E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

Interfaz Python para la librería C++ lib_3d_mec_ginac

Grado en Ingeniería Informática

Trabajo Fin de Grado

Víctor Ruiz Gómez

Pamplona, 30 de mayo de 2020

upna Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Dedico este proyecto a mi familia por el apoyo moral y económico que me han dado antes, durante y después de la realización del mismo.

Contentidos

1 Resumen **1**
- 1.1 Traducción al inglés . . . 1
- 1.2 Palabras clave . . . 2

2 Introducción **3**
- 2.1 Objetivos . . . 3
  - 2.1.1 Objetivo principal . . . 3
  - 2.1.2 Objetivos secundarios . . . 4
- 2.2 Tecnologías y herramientas software empleadas . . . 4
  - 2.2.1 Lenguajes de programación . . . 4
  - 2.2.2 Frameworks y librerías . . . 4
  - 2.2.3 Servicios . . . 5
- 2.3 Metodologías de desarrollo software empleadas . . . 5
  - 2.3.1 Fases del proyecto . . . 5
- 2.4 La librería lib_3d_mec_ginac . . . 6
  - 2.4.1 Ejemplo de uso . . . 7

3 Requisitos de usuario **11**
- 3.1 La interfaz Python . . . 11
- 3.2 Entorno gráfico . . . 13
  - 3.2.1 Diseño del visor 3D . . . 14
  - 3.2.2 Diseño de la interfaz de usuario . . . 14
  - 3.2.3 Interfaz por consola de comandos . . . 15

4 Diseño e implementación del software **17**
- 4.1 Arquitectura del software . . . 17
  - 4.1.1 El núcleo de la librería . . . 18
  - 4.1.2 Módulo de renderización de gráficos o «drawing» . . . 22
  - 4.1.3 El módulo de interfaz de usuario . . . 30
- 4.2 API funcional de la librería . . . 31

5 Pruebas de software **33**

6 Documentación del proyecto **35**
- 6.1 Referencia de la API . . . 35

7 Publicación del software **39**

7.1 El código fuente . . . 39
7.2 Imágenes docker . . . 39
7.3 Servicios en la nube . . . 39
7.4 Licencia . . . 40

8 Conclusiones y líneas futuras **41**

9 Bibliografía **43**

10 Apéndice **45**
10.1 Operaciones aritméticas entre objetos . . . 45

CHAPTER 1

Resumen

«lib_3d_mec_ginac» es una librería de código implementada en el lenguaje de programación C++, que sirve como herramienta para el diseño y estudio de sistemas mecánicos multicuerpo. Permite el modelado de sistemas mediante la escritura de programas codificados en C++ para obtener las ecuaciones de la cinemática y dinámica, que pueden emplearse para analizar el comportamiento de estos mecanismos.

Este proyecto tiene como objetivo trasladar las funcionalidades de «lib_3d_mec_ginac» al lenguaje de alto nivel e interpretado Python para mejorar la interacción entre el usuario final y la librería.

Se desarrolla además un entorno gráfico que permite visualizar y realizar simulaciones en 3D de los sistemas modelados con este software.

1.1 Traducción al inglés

«lib_3d_mec_ginac» is a source code library implemented in the C++ progamming language, used as a tool for the design and study of multibody mechanical systems. It allows the modeling of systems by writting a program in C++ in order to obtain the kinematic and dynamic equations which can be used to analyze the behaviour of those mechanisms.

The goal of this project is to shift the features of «lib_3d_mec_ginac» to the high level interpreted programming language Python in order to improve the interaction between the user and the library.

A 3D graphical environment is also developed to allow the visualization and simulation of systems modeled with this software.

1.2 Palabras clave

Python, C++, Cython, «language binding», «lib_3d_mec_ginac», «multibody dynamics», «3D simulation», «3D visualization»

CHAPTER 2

Introducción

2.1 Objetivos

2.1.1 Objetivo principal

El objetivo principal de este proyecto es dotar a la librería «lib3d-mec-ginac» de una mayor usabilidad y proveer un entorno programático dónde sea posible acceder a todas sus funcionalidades de una forma más sencilla.

En otras palabras, se desea añadir una capa adicional de software sobre la librería que cubra sus carencias en cuanto a la interacción con el usuario, que son principalmente las siguientes:

- Al ser una librería «standalone», es decir, un paquete de código, debe codificarse una programa en el mismo lenguaje de programación sobre el cual está desarrollado (C++) y emplear la API expuesta (un conjunto de rutinas y clases) para hacer uso de la misma. Esto obliga al usuario a emplear ciertos formalismos lingüísticos y una sintaxis que con frecuencia dificulta la legibilidad y escritura de los programas.
- C++ obliga a que un programa deba compilarse antes de poder ser ejecutado, es decir, debe convertirse en un archivo binario (empleando para ello un compilador como g++ o gcc)

  Además, cualquier cambio introducido en el código por mínimo que sea, obligará a repetir de nuevo esta tarea. Por ello, el programa no responde de forma inmediata antes las modificaciones introducidas en el código fuente, haciendo que la interacción usuario-máquina no pueda producirse en tiempo real.

La solución que se propone para resolver estos problemas de usabilidad consiste en trasladar las funcionalidades de la librería a otro lenguaje de programación de más alto nivel que C++ y que posea las siguientes cualidades:

- Debe ser un lenguaje interpretado. El código deberá ser ejecutado con la ayuda de un intérprete de comandos, eliminando de este modo la fase de compilación.
- Fácil de utilizar y con una curva de aprendizaje baja. Además debe proveer una sintaxis sencilla y más flexible que C++
- Tiene que ser conocido en el entorno académico en general y en el área de la ingeniería

En definitiva, se quiere operar con «lib_3d_mec_ginac» desde un lenguaje que cumpla esta lista de requerimientos pero sin perder eficiencia computacional a la hora de ejecutar sus algoritmos.

Python es lenguaje objetivo escogido para este propósito y la estrategia adoptada para instanciar la solución propuesta es la elaboración de una interfaz lógica o «language binding» que comunique «lib_3d_mec_ginac» con este lenguaje. El desarrollo de esta no consiste una traducción literal del código de la librería a Python, sino que es un software adicional que actúa de nexo entre «lib_3d_mec_ginac» y Python.

2.1.2 Objetivos secundarios

Como objetivo adicional del proyecto se propone la elaboración de un entorno gráfico integrado con la interfaz, que permita visualizar y realizar simulaciones en 3D interactivas de los sistemas mecánicos modelados, tomando como referencia e inspiración el diseño de la herramienta «3D_MEC» para su desarrollo.

2.2 Tecnologías y herramientas software empleadas

Esta sección muestra todos los recursos software utilizados (lenguajes de programación, frameworks y herramientas)

2.2.1 Lenguajes de programación

- **C++**: Es un lenguaje de programación que surge de C. Dota a este último de capacidades y características adicionales como el paradigma orientado a objetos, el uso de plantillas genéricas o «templates» y una librería «standard» más extensa.

  Es un lenguaje útil para desarrollar aplicaciones de alto rendimiento en donde es necesario un control completo sobre los recursos empleados por los programas y una mayor eficiencia computacional.

- **Python**: Es un lenguaje de programación de alto nivel interpretado, es decir, que se ejecuta sin la necesidad de compilar el código. Soporta distintos paradigmas como el funcional, orientado a objetos, . . .

  Hoy en día es el lenguaje más utilizado para desarrollar aplicaciones en muchas ramas de la ingeniería. Destaca por su flexibilidad y facilidad de uso.

- **Cython**: Es un lenguaje de programación cuyo objetivo es facilitar la escritura de extensiones de C/C++ para Python. Tiene la misma sintaxis y reglas semánticas que Python pero además permite la invocación de funciones en C y la declaración de variables con tipos estáticos. Se usará para comunicar Python y las librería C++ «lib_3d_mec_ginac»

  Se eligió de entre varios frameworks de creación de extensiones (como «Boost.Python» o «Numba») por su sencillez en cuanto a la escritura de código.

- **reStructuredText**: Es un lenguaje de marcado que permite dar formato y estructura a un texto. El código puede ser convertido a documentos como pdf o html utilizando la herramienta Sphinx. Es utilizado para escribir la documentación técnica sobre la librería y la memoria del proyecto.

2.2.2 Frameworks y librerías

La siguiente lista muestra los paquetes de código y frameworks empleados para el desarrollo software del proyecto:

- **GiNaC**: Es un framework de computación simbólica de propósito general escrito en el lenguaje C++. «lib_3d_mec_ginac» se construye encima de esta librería.
- **NumPy**: Es un framework de computación numérica en Python. Permite representar vectores N-dimensionales. Provee un conjunto de funciones matemáticas para operar con estas.
- **VTK**: Es una librería de renderización de gráficos 3D implementada en C++ (pero posee bindings para el lenguaje Python)

- **Pytest**: Framework que facilita la escritura de tests unitarios en Python.
- **tkinter**: Librería para la creación de interfaces de usuario por defecto de Python.
- **Sphinx**: Es un generador de documentación automática que convierte ficheros con la extensión «.rst» escritos en el lenguaje de marcado «reStructuredText» a documentos pdf, html, man, . . .
- **Git**: Git es un software de control de versiones. Permite el desarrollo de software de forma colaborativa, la bifurcación del código en múltiples ramas, . . .
- **Docker**: Es una tecnología de virtualización. Ha sido empleada para realizar pruebas de despliegue del software sobre distintos sistemas operativos y entornos de producción.

2.2.3 Servicios

Servidores, páginas web y servicios online utilizados:

- **Github** y **Bitbucket** son sitios web de almacenamiento de proyectos software que utilizan el sistema de control de versiones git. Al inicio del proyecto, se usó BitBucket ya que la opción de alojamiento de proyectos privados no estaba disponible en Github de forma gratuita. Actualmente el software de este proyecto se encuentra alojado en este último.
- **Heroku**: Servicio en línea para desplegar imágenes docker de forma remota.

2.3 Metodologías de desarrollo software empleadas

SCRUM es un conjunto de buenas prácticas que facilita el desarrollo ágil del software. Ha sido empleada ya que es útil en proyectos donde los requisitos de usuario o funcionales pueden variar a lo largo del tiempo (por ejemplo, si no se conocen todas las librerías de código o herramientas que van a utilizarse a priori).

A pesar de ser un trabajo individual, se hecho necesaria la comunicación y colaboración con algunos de los desarrolladores de «lib_3d_mec_ginac» (Javier Ros y Aitor Plaza) dada la relación que tiene con este proyecto. Estos han interpretado el rol de «clientes» dentro de la metodología SCRUM, teniendo como tarea el planteamiento de los requisitos de usuario y dar «feedback» de las funcionalidades implementadas de cara a introducir nuevas mejoras y características al producto software.

2.3.1 Fases del proyecto

Este marco de trabajo planifica el desarrollo del proyecto en distintas fases o iteraciones, normalmente de corta duración, entre dos y cuatro semanas. En cada una de ellas, se crea una lista priorizada de requisitos que se acuerdan con los clientes al inicio de la misma. Luego, se elabora una lista de tareas necesarias para llevarlos a término. Las tareas pueden ser de distinta índole: De planificación, implementación, validación y verificación del software, documentación, . . .

Una vez finalizada la iteración, el producto adquiere nuevas funcionalidades tangibles (el desarrollo es iterativo e incremental). Estas se exponen a los clientes para llevar a cabo una retrospectiva del proyecto, obtener «feedback» y proponer mejoras.

El desarrollo del proyecto puede dividirse en las siguientes fases:

- **Etapa de formación**: Se finalizó en una única iteración de dos semanas. En esta se consiguió aprender a utilizar las librerías C++ «lib_3d_mec_ginac» y «GiNaC». Así mismo, fue necesario conocer los conceptos básicos y algoritmos empleados en el área de la computación simbólica. En esta etapa no se desarrolló software.

- **Fase de desarrollo del prototipo**: Se elabora la primera versión de la extensión «lib_3d_mec_ginac» a Python. No satisface todos los requisitos de usuario marcados pero sirve como prueba de concepto. Se finalizó en un ciclo de tres semanas de duración.
- **Desarrollo de la interfaz Python**: Partiendo del prototipo se implementaron las funcionalidades restantes para satisfacer todos los requisitos de usuario. Se dividió en dos iteraciones de dos semanas de duración.
- **Desarrollo del entorno gráfico**: Se elaboró el entorno gráfico 3D integrado con la interfaz. Fueron necesarias cuatro iteraciones de dos semanas.
- **Publicación del proyecto**: Durante tres semanas se documentó el proyecto, añadiendo ejemplos de uso y escribiendo la «referencia de la API» (información técnica detallada sobre el empleo de cada componente del software). Se realizaron pruebas de despliegue y se hizo público el código del proyecto.

Después de la última fase, el desarrollo del software sigue activo; Debe mantenerse en una continua revisión para corregir errores en el código.

2.4 La librería lib_3d_mec_ginac

Esta sección muestra brevemente como puede emplearse «lib_3d_mec_ginac». La librería la creación de un conjunto de primitivas geométricas cuyas propiedades se parametrizan en función de una serie de símbolos (variables matemáticas). Estas primitivas son ensambladas para definir los componentes del sistema y las uniones entre estos.

Existen distintos tipos de primitivas:

- **Base de proyección**: Es una base ortonormal representada en el espacio 3-dimensional.
- **Vector**: Representa una entidad con dirección, sentido y módulo en el espacio. Además están proyectadas siempre sobre una base.
- **Punto**: Son posiciones concretas del espacio. Se definen usando un vector y sus coordenadas son relativas a un «punto origen».
- **Marco de referencia o "Frame"**: Es una base de proyección que puede posicionarse en un punto concreto del espacio.
- **Sólido**: Es un frame que representa además una entidad con masa y tensor de inercia. Esta es usada para el planteamiento de ecuaciones dinámicas.

Y existen distintas clases de símbolos. Algunos son los siguientes:

- **Parámetros**: Se utilizan para representar variables geométricas o físicas del sistema (por ejemplo la la gravedad)
- **Coordenadas generalizadas**: Pueden representar de forma abstracta distintos tipos de magnitudes en función del formalismo empleado.

  Por ejemplo, la posición de una partícula sobre un eje específico del espacio euclídeo 3D, la magnitud en radianes representando un rotación sobre un eje.

  Tienen asociadas otras dos variables que representan sus dos primeras derivadas (velocidad y aceleración)
- **Tiempo**: Es una variable que representa la dimensión temporal.

Cada tipo de símbolo y primitiva puede ser creada mediante una rutina específica de la API, como se muestra en el *Ejemplo de uso*

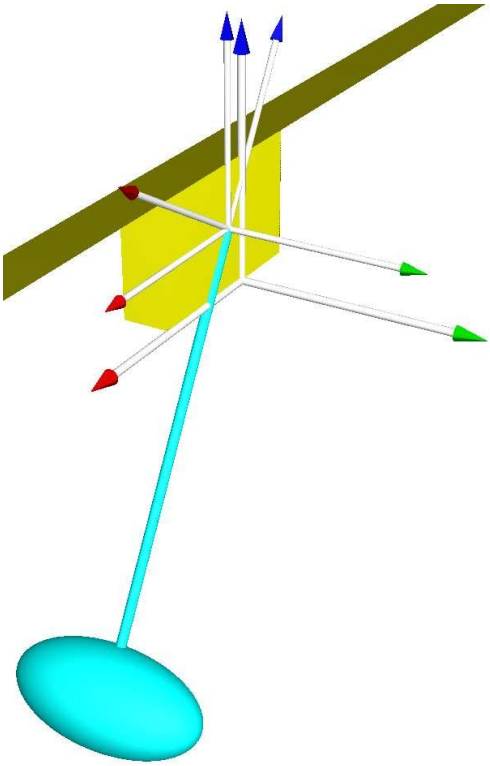
Las propiedades geométricas o físicas de las primitivas (como por ejemplo las componentes de un vector sobre cada eje del espacio) se definen empleando símbolos o expresiones simbólicas (formulas matemáticas en donde aparecen una o más variables unidas mediante operadores aritméticos o funciones).

A partir de estas primitivas, se plantean las restricciones del sistema. Estas permitirán definir el conjunto de todos los posibles valores numéricos para las variables definidas que hagan que el sistema sea «coherente». Cada restricción se plantea como una ecuación diferencial que relaciona varios parámetros y coordenadas del sistema. Estas son representadas como expresiones simbólicas.

2.4.1 Ejemplo de uso

Este ejemplo ha sido extraído del «paper» de lib_3d_mec_ginac (ver la *Bibliografía*). Crea un sistema mecánico sencillo compuesto por los siguientes componentes:

- Un bloque que se desplaza sobre el eje X (pintado de rojo en la figura)
- Un péndulo que gira con respecto al eje Y (en verde).



Las ecuaciones de restricción del mecanismo son obtenidas usando el formalismo «Newton-Euler».

```
#include "Matrix.h"
#include "Vector3D.h"
#include "Tensor3D.h"
#include "System.h"

#include <ginac/ginac.h>
using namespace GiNaC;

int main(){
    //System definition
    System sys;
    atomize=0;

    //Coordinate definition
```

(continué en la próxima página)

(proviene de la página anterior)

```
    symbol x = *sys.new_Coordinate("x");
    symbol theta = *sys.new_Coordinate("theta");

    //Kinematical parameter definition
    symbol d = *sys.new_Parameter("d");
    symbol l = *sys.new_Parameter("l");
    symbol r = *sys.new_Parameter("r");

    //Define Base
    sys.new_Base("BPendulum","xyz",0,1,0, -theta);

    //Define
    Vector3D OGb = *sys.new_Vector3D("OGb",x,0,d/2,"xyz");
    Vector3D OA =  *sys.new_Vector3D("OA",x,0,0,"xyz");
    Vector3D GbGp= *sys.new_Vector3D("GbGp",0,0, -l,"BPendulum");

    //Define Points
    Point* Gb = sys.new_Point("Gb","O",&OGb);
    Point* A = sys.new_Point("A","O",&OA);
    Point* Gp = sys.new_Point("Gp","Gb",&GbGp);

    //Define Frames
    Frame* Block    = sys.new_Frame("Block","Gb","xyz");
    Frame* Pendulum = sys.new_Frame("Pendulum",Gp,"BPendulum");

    //Dynamical Parameter Definition
    symbol mblock =    *sys.new_Parameter("mblock");
    symbol mpendulum = *sys.new_Parameter("mpendulum");
    symbol g =         *sys.new_Parameter("g");
    symbol I1 =        *sys.new_Parameter("I1");
    symbol I2 =        *sys.new_Parameter("I2");
    symbol I3 =        *sys.new_Parameter("I3");
    Tensor3D IpendulumGp = *sys.new_Tensor3D("IpendulumGp",
        I1,0,0,0,I2,0,0,0,I3,"BPendulum");

    // Joint unknowns definition
    symbol Fgby = *sys.new_Joint_Unknown("Fgby");
    symbol Fgbz = *sys.new_Joint_Unknown("Fgbz");
    symbol Mgbx = *sys.new_Joint_Unknown("Mgbx");
    symbol Mgby = *sys.new_Joint_Unknown("Mgby");
    symbol Mgbz = *sys.new_Joint_Unknown("Mgbz");
    symbol Fbpx = *sys.new_Joint_Unknown("Fbpx");

    symbol Fbpy = *sys.new_Joint_Unknown("Fbpy");
    symbol Fbpz = *sys.new_Joint_Unknown("Fbpy");
    symbol Mbpx = *sys.new_Joint_Unknown("Mbpx");
    symbol Mbpz = *sys.new_Joint_Unknown("Mbpz");

    //Joint Torsor Definition
    Vector3D F_GB    = *sys.new_Vector3D("F_GB",0,Fgby,Fgbz,"xyz");
    Vector3D M_GB_A  = *sys.new_Vector3D("M_GB_A",Mgbx,Mgby,Mgbz,"xyz");
    Vector3D F_BP    = *sys.new_Vector3D("F_BP",Fbpx,Fbpy,Fbpz,"xyz");
    Vector3D M_BP_Gb = *sys.new_Vector3D("M_BP_Gb",Mbpx,0,Mbpz,"xyz");

    //Constitutive Forces and moments Definition
    Vector3D Block_gravity =    *sys.new_Vector3D("mblock*g",
        0,0,-mblock * g,"xyz");
```

(continué en la próxima página)

(proviene de la página anterior)

```
    Vector3D Pendulum_gravity = *sys.new_Vector3D("mpendulum*g",
        0,0,-mpendulum * g,"xyz");

    //Define Velocity and Acceleration of Point Gb
    Vector3D VabsGb = sys.Dt(OGb,"abs");
    Vector3D AabsGb = sys.Dt(VabsGb,"abs");

    //Define Acceleration of Point Gp
    Vector3D OGp = sys.PositionVector("O","Gp");
    Vector3D VabsGp = sys.Dt(OGp,"abs");
    Vector3D AabsGp = sys.Dt(VabsGp,"abs");

    //Angular moment and Inertia Moment of pendulum
    Vector3D OmegaPendulum = sys.AngularVelocity("xyz","BPendulum");
    Vector3D H_Gp = IpendulumGp * OmegaPendulum;
    Vector3D Iner_Moment_Pendulum_Gp = -sys.Dt(H_Gp,"xyz");

    //Dynamic Equations
    Matrix equation_1_3 = -mblock * AabsGb + Block_gravity
        + F_GB - F_BP;
    Matrix equation_2_3 = ((sys.PositionVector("Gb","A") ^ F_GB) + M_GB_A)
        - M_BP_Gb;
    Matrix equation_3_3 = -mpendulum * AabsGp + Pendulum_gravity + F_BP;
    Matrix equation_4_3 = Iner_Moment_Pendulum_Gp
        + ((sys.PositionVector("Gp","Gb") ^ F_BP) + M   _BP_Gb);

    Matrix Dynamic_Equations(4,1,
    &equation_1_3,&equation_2_3,&equation_3_3,&equation_4_3);

    Matrix q       = sys.Coordinates();
    Matrix dq      = sys.Velocities();
    Matrix ddq     = sys.Accelerations();
    Matrix epsilon = sys.Joint_Unknowns();

    Matrix M = Jacobian(Dynamic_Equations.transpose(),ddq);

    Matrix V = Jacobian(Dynamic_Equations.transpose(),epsilon);
    Matrix Q = Dynamic_Equations-(M*ddq+V*epsilon);

    // Print dynamic equations
    cout << Dynamic_Equations << endl;
}
```

Las ecuaciones son las siguientes (cada elemento muestra la parte izquierda de la ecuación que se iguala a cero):

```
[-Fbpx-ddx*mblock,
 -Fbpy+Fgby,
 -Fbpz+Fgbz-g*mblock,
  Mgbx+1/2*d*Fgby,
  Mgby,
  Mgbz,
  Fbpx-ddx*mpendulum,
  Fbpy,
  Fbpz-g*mpendulum,
  0,
  ddtheta*I2,
  0]
```

CHAPTER 3

Requisitos de usuario

3.1 La interfaz Python

La interfaz a desarrollar será un paquete de código que podrá ser empleado en un script Python. Sus elementos (funciones y clases) expuestos podrán ser importados durante la ejecución del programa utilizando la sintaxis habitual del lenguaje (instrucción `import`):

```
from lib3d_mec_ginac import *
```

Las rutinas y clases implementadas en la librería «lib_3d_mec_ginac» en C++ serán las mismas que las expuestas por el paquete Python en cuanto a funcionalidad y forma de uso.

El código Python que se muestra a continuación deberá proporcionar el mismo resultado que el programa C++ presentado en la sección *Ejemplo de uso*

```
from lib3d_mec_ginac import *

# System definition
sys = System()
disable_atomization()

# Coordinate definition
x = sys.new_coordinate('x')[0]
theta = sys.new_coordinate('theta')[0]

# Kinematical parameter definition
d = sys.new_parameter('d')
l = sys.new_parameter('l')
r = sys.new_parameter('r')

# Define base
sys.new_base('BPendulum', 'xyz', 0, 1, 0, -theta)

# Define vectors
```

(continué en la próxima página)

(proviene de la página anterior)

```
OGb =  sys.new_vector('OGb',  x, 0, d/2, "xyz")
OA  =  sys.new_vector('OA',   [x, 0, 0],  "xyz")
GbGp = sys.new_vector('GbGp', 0, 0, -1, base='BPendulum' )

# Define points
Gb = sys.new_point("Gb", "O",  OGb)
A  = sys.new_point("A",  "O",  OA)
Gp = sys.new_point("Gp", "Gb", GbGp)

# Define frames
Block =    sys.new_frame("Block",    "Gb", "xyz")
Pendulum = sys.new_frame("Pendulum",  Gp,  "BPendulum")

# Dynamical parameter definition
mblock =      sys.new_parameter("mblock")
mpendulum =   sys.new_parameter("mpendulum")
g =           sys.new_parameter("g")
I1, I2, I3 =  [sys.new_parameter(name) for name in ('I1', 'I2', 'I3')]
IpendulumGp = sys.new_tensor("IpendulumGp", I1, 0, 0, 0, I2, 0, 0, 0, I3, "BPendulum")

# Joint unknown definitions
Fgby, Fgbz       = map(sys.new_joint_unknown, ['Fgby', 'Fgbz'])
Mgbx, Mgby, Mgbz = map(sys.new_joint_unknown, ['Mgbx', 'Mgby', 'Mgbz'])
Fbpx, Fbpy, Fbpz = map(sys.new_joint_unknown, ['Fbpx', 'Fbpy', 'Fbpz'])
Mbpx, Mbpz       = map(sys.new_joint_unknown, ['Mbpx', 'Mbpz'])

# Joint torsor definition
F_GB    = sys.new_vector('F_GB',   0, Fgby, Fgbz, "xyz")
M_GB_A  = sys.new_vector('M_GB_A', Mgbx, Mgby, Mgbz, "xyz")
F_BP    = sys.new_vector('F_BP',   Fbpx, Fbpy, Fbpz, "xyz")
M_BP_Gb = sys.new_vector('M_BP_Gb', Mbpx, 0, Mbpz, "xyz")

# Constitutive forces and moments definition
Block_gravity = sys.new_vector('mblock_g', 0, 0, -mblock * g, "xyz")
Pendulum_gravity = sys.new_vector('mpendulum_g', 0, 0, -mpendulum*g, "xyz")

# Define Velocity and Acceleration of Point Gb
VabsGb = sys.dt(OGb, "abs")
AabsGb = sys.dt(VabsGb, "abs")
OGp = sys.position_vector("O", "Gp")
VabsGp = sys.dt(OGb, "abs")
AabsGp = sys.dt(VabsGp, "abs")

# Angular moment and Inertia Moment of pendulum
OmegaPendulum = sys.angular_velocity("xyz", "BPendulum")
H_Gp = IpendulumGp * OmegaPendulum
Iner_Moment_Pendulum_Gp = -sys.dt(H_Gp, "xyz")

# Dynamic equations
Dynamic_Equations = Matrix.block(4, 1,
    -mblock * AabsGb + Block_gravity + F_GB - F_BP,
    (cross(sys.position_vector("Gb","A"), F_GB) + M_GB_A),
    -mpendulum * AabsGp + Pendulum_gravity + F_BP,
    Iner_Moment_Pendulum_Gp
)

q =       sys.get_coordinates_matrix()
```

(continué en la próxima página)

(proviene de la página anterior)

```
dq =      sys.get_velocities_matrix()
ddq =     sys.get_accelerations_matrix()
epsilon = sys.get_joint_unknowns_matrix()

M = sys.jacobian(Dynamic_Equations.transpose(), ddq)
V = sys.jacobian(Dynamic_Equations.transpose(), epsilon)
Q = Dynamic_Equations - (M * ddq + V * epsilon)
print(Q)
```

Se desea que los progrmas escritos en C++ puedan traducirse a Python sin llevar a cabo demasiadas modificaciones, tal y como ocurre en el ejemplo mostrado. Por lo tanto, los nombres de las funciones y clases en la API de Python deberán ser lo más parecidos a los empleados en C++.

De forma adicional, se deberá poder aprovechar las capacidades que brinda el lenguaje Python para obtener una mayor flexibilidad en la escritura y claridad del código:

- El tipado estático no es necesario por lo que las variables no se declaran junto a un tipo específico. De esta forma no es necesario recordar el nombre de las clases de los valores devueltos por las rutinas de la API.
- Algunos métodos C++ devuelven punteros mientras que otros proporcionan el resultado «por valor». Python solo maneja los objetos «por referencia» (no existe el concepto de puntero a dato). Con esto se consigue que las llamadas a rutinas de la API sean más sencillas de escribir.
- Python proporciona además un conjunto de funciones predefinidas y una sintaxis propicia para reducir al máximo el código redundante y compactarlo sin perder claridad. Por ejemplo, utilizando listas y la función `map` como se muestra en el ejemplo.
- La posibilidad de pasar argumentos de forma posicional o por palabra clave, permite una mayor flexibilidad en cuanto a la invocación de funciones de la API

3.2 Entorno gráfico

Para llevar a término el objetivo secundario, debe desarrollarse un entorno gráfico en el lenguaje Python que permita la visualización de los sistemas mecánicos modelados con la librería. Debe cumplir con los siguientes requerimientos:

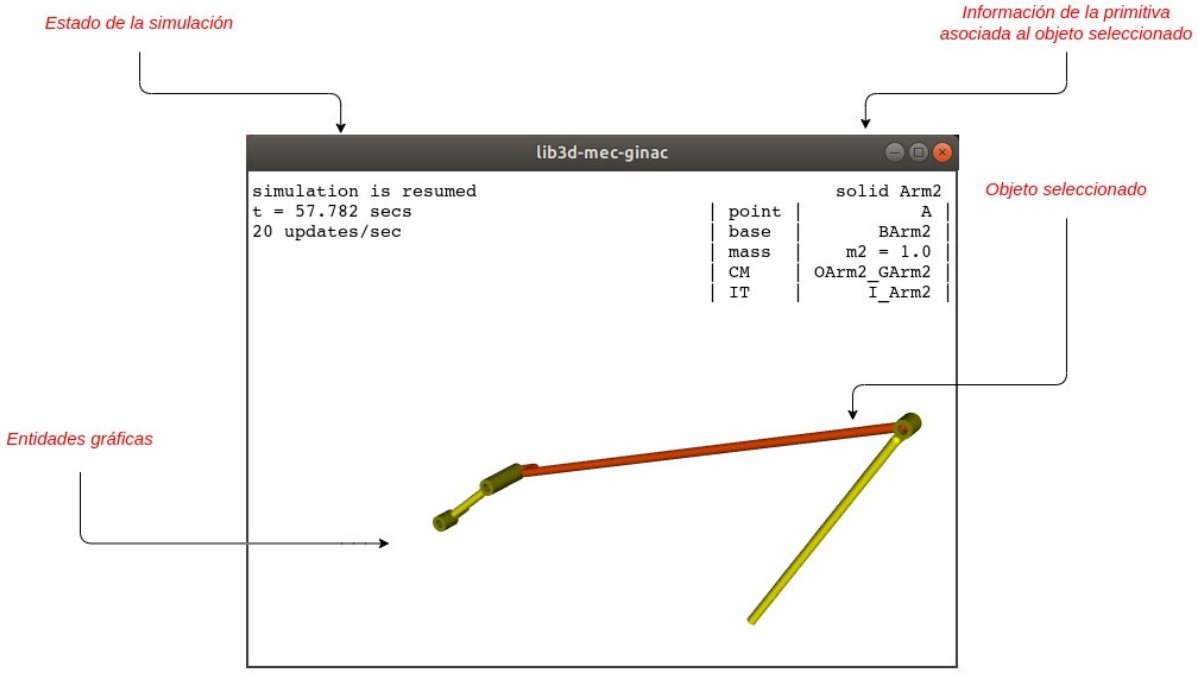
- Las primitivas punto, frame, vector y sólido pueden ser dibujadas en el visor invocando de manera explicita un conjunto de rutinas expuestas en la API de la interfaz Python.
- Podrá visualizarse un sistema mecánico de forma interactiva, permitiendo al usuario controlar la cámara empleada para enfocar la escena 3D, empleando los dispositivos de entrada (teclado y ratón) para modificar su posición, rotación, . . . Además, podrá seleccionar las primitivas haciendo click en estas para obtener información descriptiva de las mismas (por ejemplo, para obtener las componentes o el nombre de la base de un vector).
- El sistema mecánico además podrá ser simulado realizando una animación sobre las entidades gráficas dibujadas.
- Deberá añadirse una interfaz de usuario gráfica provista de una serie de menús contextuales que permiten controlar los parámetros y el estado de la simulación del mecanismo y modificar la visibilidad de las primitivas renderizadas. Finalmente, el usuario deberá ser capaz de ejecutar instrucciones Python en una consola interactiva integrada en la propia interfaz gráfica.

3.2.1 Diseño del visor 3D

El visor 3D deberá estar compuesto por los siguientes elementos gráficos:

- Un texto que muestra el estado de la simulación (si está en ejecución o en pausa, el tiempo transcurrido y el número de pasos de simulación realizados por segundo).
- La primitiva seleccionada por el usuario aparecerá en la pantalla resaltada (en color rojo) e información relevante de la misma será imprimida en forma de texto.

La siguiente figura muestra el diseño que debe tener el visor 3D.

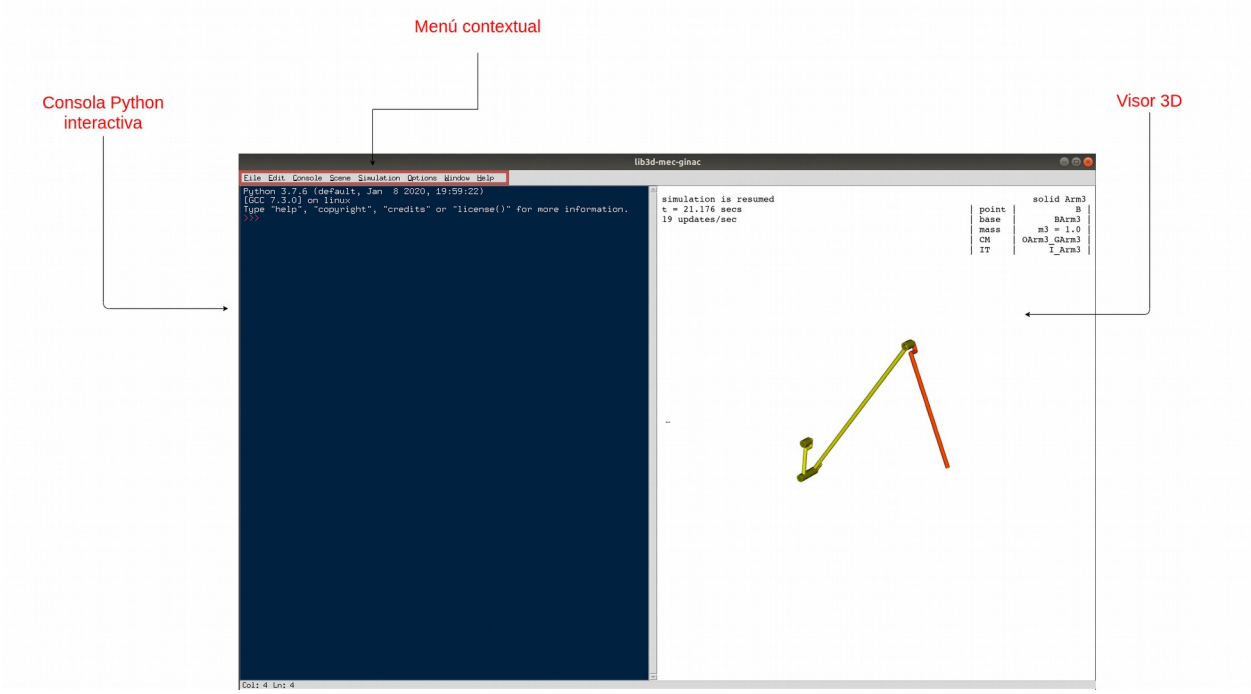


3.2.2 Diseño de la interfaz de usuario

La interfaz de usuario es el visor 3D con elementos gráficos adicionales:

- Menús contextuales:
  - El menú *Simulaton* tendrá opciones para parar/iniciar y pausar/resumir la simulación, además de distintos submenús que permitirán modificar los distintos parámetros de la misma.
  - El menú *Scene* permite modificar la visibilidad de los elementos gráficos (mostrarlos u ocultarlos).
- Una consola Python: Emula una sesión Python interactiva que permite ejecutar cualquier instrucción Python. Pueden emplearse todas las rutinas de la API dentro de está sesión Python. Es posible entonces modificar la visibilidad de la escena 3D, los valores de los parámetros del sistema mecánico, . . . El entorno gráfico se actualizará de forma inmediata en respuesta al código ejecutado en la consola.

El diseño deberá ser el siguiente:



3.2.3 Interfaz por consola de comandos

Por última deberá implementarse un programa que permita ejecutar un script python y abrir el entorno gráfico utilizando un único comando en una sesión bash de Linux.

El programa podrá invocarse del siguiente modo (una vez finalizado el proceso de instalación de la librería)

```
$ python -m lib3d_mec_ginac [ruta del script]
```

y deberá aceptar los siguientes argumentos (opcionales):

- «–help» (imprime la página de ayuda)
- «–no-gui» (ejecuta el script sin abrir el entorno gráfico)
- «–no-ide» (ejecuta el script y abre el entorno gráfico pero elimina los menus contextuales y la consola python mostrando de este modo unicamente el visor 3D)

CHAPTER 4

Diseño e implementación del software

En esta sección se hablará sobre que patrones de diseño se han empleado para la elaboración del software, además de explicar brevemente como se ha llevado a cabo su implementación.

4.1 Arquitectura del software

Se ha realizado una división del software en distintos módulos o componentes lógicos. Los más relevantes son los siguientes:

- El núcleo de la librería o también «core», trae de C++ las funcionalidades de los frameworks GiNaC y lib_3d_mec_ginac. Además implementa algoritmos para la resolución analítica de las ecuaciones de restricción (métodos de integración numérica)

  Es el módulo principal y el más relevante de la librería. Fue el primero en ser implementado y satisface los requisitos de usuario definidos para alcanzar la meta principal de este proyecto (elaboración de una interfaz Python a «lib_3d_mec_ginac»).

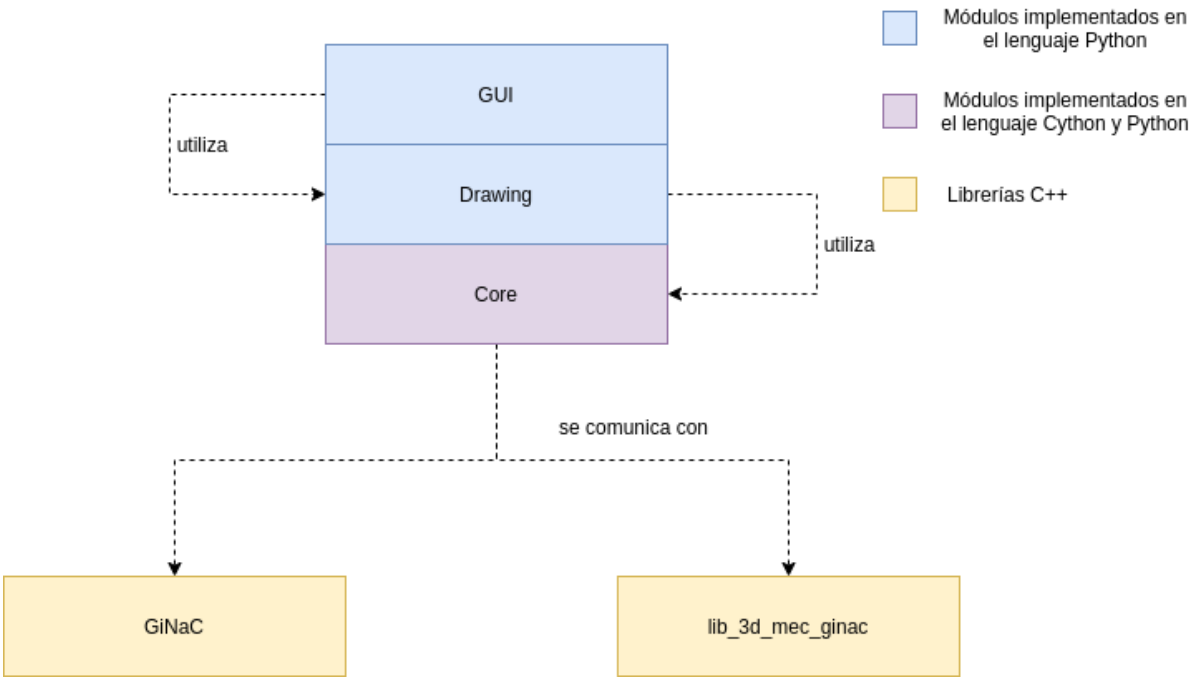
  Está escrito en los lenguajes Cython y Python. Queda diseñado de tal modo que pueda operar sin depender del resto de módulos creados.

- Módulo de renderización de gráficos o «drawing»: Su función es proveer un conjunto de rutinas que permiten dibujar y simular los sistemas mecánicos elaborados mediante las primitivas geométricas de lib_3d_mec_ginac (usando el paquete «core»), sobre un entorno gráfico 3D.

  Este módulo esta implementado enteramente en el lenguaje Python y hace uso de las librerías de gráficos VTK y OpenGL.

- Módulo de interfaz gráfica: Añade una interfaz de usuario (elementos interactivos como los menús contextuales) al visor 3D.

Los dos últimos módulos se han desarrollado para alcanzar los objetivos secundarios del proyecto.



Cada uno de los módulos puede verse como un «incremento» o «iteración» del proyecto, añadiendo de este modo funcionalidades extra a la librería. La implementación de cada uno de ellos depende de los módulos creados previamente y puede operar con independencia del resto de módulos posteriores.

A continuación se detalla la estructura interna, diseño e implementación de cada uno de ellos.

4.1.1 El núcleo de la librería

Este módulo esta dividido en dos componentes lógicos:

- Una «extensión Cython» que se encarga de gestionar la comunicación entre Python y las librerías de C++ GiNaC y lib_3d_mec_ginac. Traslada alguna de las rutinas y clases de estas librerías al lenguaje Python.
- Un script Python que contiene la implementación de un algoritmo para la resolución analítica de ecuaciones de restricción.

La extensión Cython

Cython es un lenguaje cuya sintaxis es casi idéntica a Python pero permite además insertar instrucciones del lenguaje C/C++. Nos referimos con «extensión» a una pieza de software que no está implementada en el lenguaje Python pero que puede embeberse dentro de este mismo lenguaje, exponiendo de este modo una interfaz programática (un conjunto de funciones y/o clases).

A menudo estas extensiones suelen desarrollarse en lenguajes como C o C++ para llevar a cabo tareas del sistema operativo de bajo nivel o para crear algoritmos que requieren de una mayor eficiencia computacional y que puedan ser utilizados en Python.

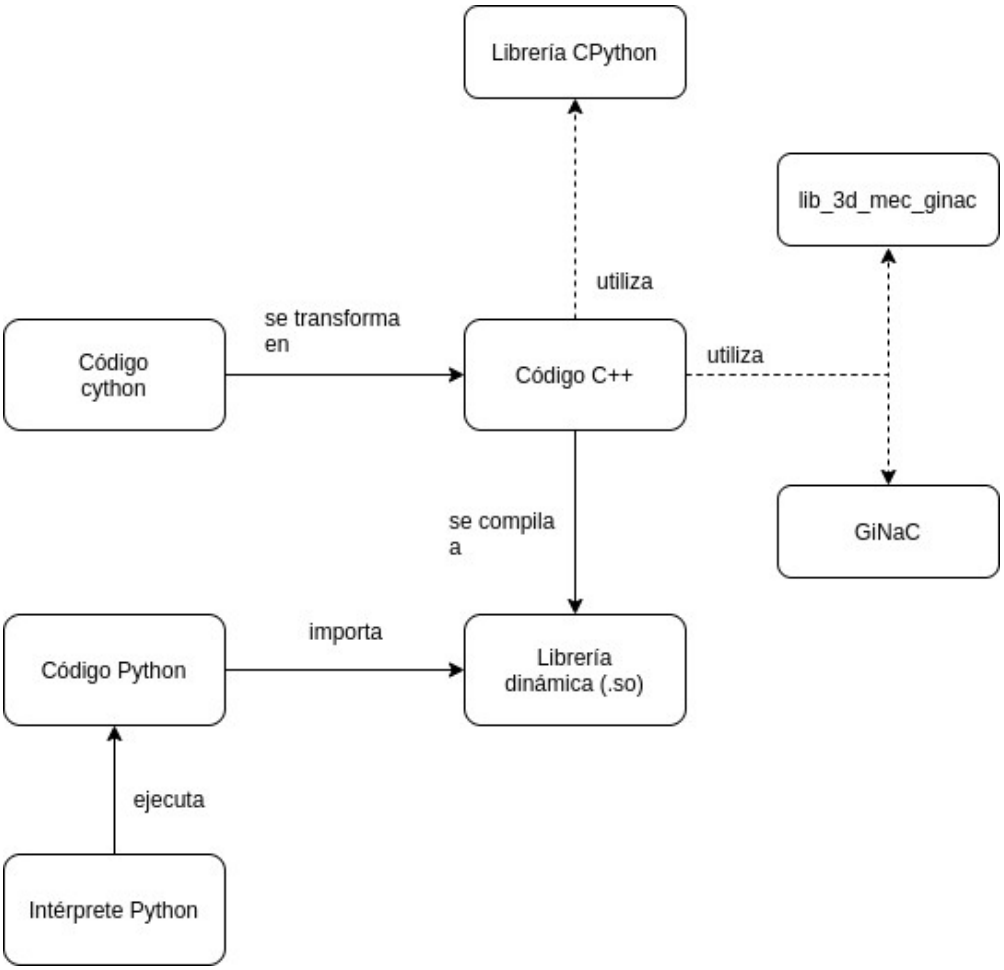
Pero para este proyecto se implementará una para desarrollar un «language binding»; Aplicación que conecta la API de un software («lib_3d_mec_ginac») con un lenguaje de programación foráneo (en este caso Python).

La implementación de la extensión se distribuye en distintos ficheros de código fuente que pueden ser agrupados en dos categorías:

- Los ficheros de declaración, en donde se indican las clases y rutinas de C++ que van a ser utilizadas e invocadas desde el intérprete Python. Son archivos de código fuente que aparecen en el repositorio con la extensión «.pxd»,
- Los ficheros de definición, que se escriben utilizando una mezcla de la sintaxis y semántica de los lenguajes Python y C++ (lenguaje Cython), e implementan las rutinas en Python en donde es necesario realizar llamadas a funciones o interactuar con objetos de C++. Estos tiene la extensión «.pyx».

Durante la fase de instalación de la librería, el código Cython se convierte a una extensión, que en última instancia puede ser empleada como un paquete Python normal. Este proceso de conversión consta de varias fases:

- El código Cython presente en los ficheros de definiciones es convertido a código fuente en C++ mediante un proceso denominado «cythonización»; Las funciones definidas en la extensión con la sintaxis Cython se convierten en rutinas escritas en el lenguaje C/C++ y emplean una serie de procedimientos de la librería CPython (la implementación por defecto de Python en C) para hacer interfaz con el intérprete.
- El código C++ generado en el paso previo, es convertido a un librería dinámica (archivo .so en Linux), utilizando un compilador C++ (g++, gcc, . . .)
- La librería dinámica es distribuida como un paquete o módulo en Python. El intérprete puede importar las funciones y clases de la extensión.

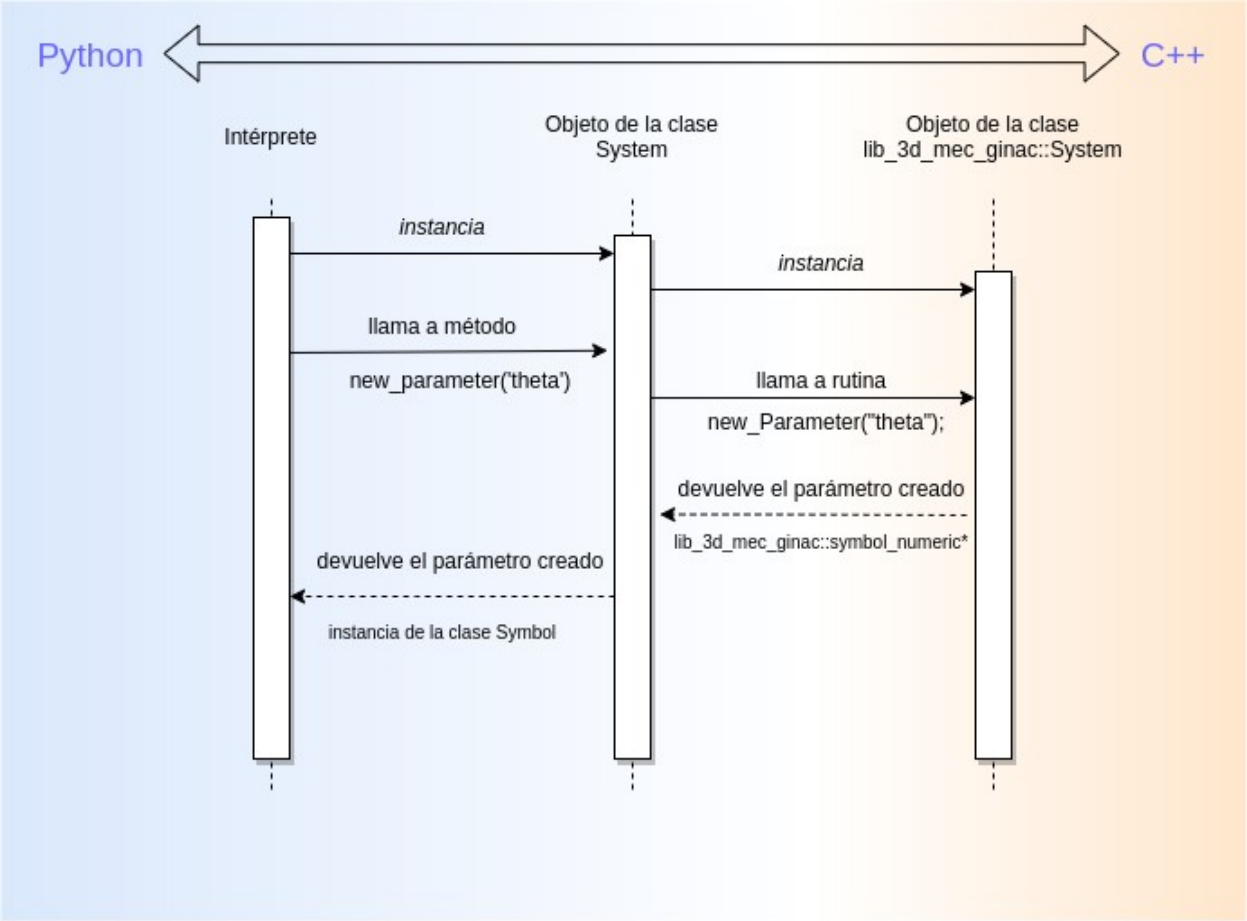


Comunicación entre Python y C++

La comunicación entre sendos lenguajes puede programarse utilizando el lenguaje Cython, el cual permite declarar funciones especiales en los ficheros de definición que pueden entremezclar código C++ y Python. Esto permite hacer llamadas a librerías externas como «GiNaC» y «lib_3d_mec_ginac» C++ y operar con objetos de clases definidas en este lenguaje.

La extensión declara un conjunto de clases. Cada una de ellas es un representante, encapsulación o «wrapper» de una clase específica de las librerías C++.

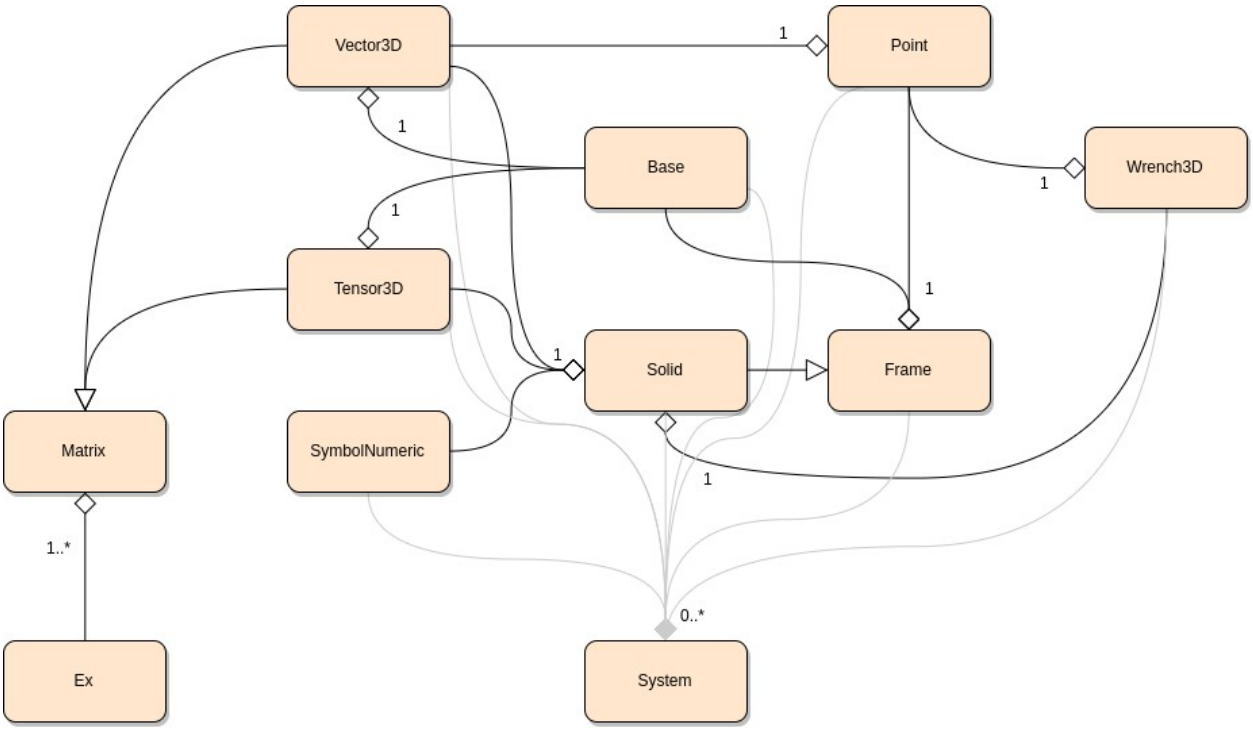
Un objeto de cualquiera de estas clases delega todas sus funcionalidades (invocaciones a métodos) a una instancia de la clase específica en C++ a la cual esté asociada. En definitiva los objetos de las clases en la parte de Python actúan como proxies u objetos intermediarios de las instancias de las clases en C++.



Las siguientes clases han sido definidas en la extensión (se muestran junto a las clases equivalentes en C++):

Clase Python	Clase C++
Ex	GiNaC::ex
SymbolNumeric	lib_3d_mec_ginac::symbol_numeric
Base	lib_3d_mec_ginac::Base
Frame	lib_3d_mec_ginac::Frame
Point	lib_3d_mec_ginac::Point
Vector3D	lib_3d_mec_ginac::Vector3D
Tensor3D	lib_3d_mec_ginac::Tensor3D
Wrench3D	lib_3d_mec_ginac::Wrench3D
Solid	lib_3d_mec_ginac::Solid
System	lib_3d_mec_ginac::System

Ya que los métodos en Python delegan las llamadas a los métodos C++, el comportamiento dinámico y estado de un objeto es el mismo en sendos lenguajes. Por ello, las relaciones de dependencia (agregaciones, composiciones, herencia) entre clases también son idénticas. Se muestran en el siguiente diagrama:



Algunas de las clases además implementan un subconjunto de operaciones del álgebra clásica como la suma, el producto, o la resta; Los objetos de las clases «symbol_numeric», «Wrench3D», «Vector3D», «Tensor3D» y «Matrix» pueden aparecer como operandos en expresiones matemáticas.

Para implementar las operaciones matemáticas en C++, se hace uso de la «sobrecarga de operadores». Esta característica permite ejecutar una función determinada pasando como argumentos los valores que aparecen en una operación matemática y el valor de retorno de esta rutina se toma como el resultado de la operación.

En Python, esto es emulado mediante la definición de unas funciones especiales denominadas «metamétodos» que son invocadas por el intérprete cuando este ejecuta operaciones aritméticas. Sobre sendos lenguajes se implementan las mismas operaciones y estas proporcionan resultados equivalentes cuando se suministran operandos idénticos. En la apéndice se adjunta información relativa a las operaciones aritméticas implementadas.

En definitiva la extensión es responsable de traer a Python un conjunto de clases de «GiNaC» y «lib_3d_mec_ginac».

Requisitos funcionales de la extensión

La extensión no implementa ninguna lógica adicional ya que solo se encarga de proporcionar una interfaz a las clases de C++, pero si que deben satisfacer ciertos requisitos a nivel funcional:

- Los métodos definidos en la extensión conectan código Python y C++. Esto significa que los argumentos de entrada deben ser convertidos a tipos de datos que el lenguaje C++ pueda entender. Del mismo modo, los valores de retorno de las rutinas C++ invocadas, deben ser transformados en objetos Python de forma consistente.

  En definitiva deben realizarse toda las conversiones pertinentes entre variables Python y C++ para que la comunicación entre ambos lenguajes sea posible.

- C++ posee características no presentes en Python como la sobrecarga de funciones, las plantillas o «templates» para la definición de rutinas, . . . La extensión debe tener en cuenta las diferencias en cuanto a los paradigmas de programación soportados.

- La sobrecarga de funciones permite establecer distintos flujos de ejecución dependiendo del tipo de los argumentos de entrada en C++. La extensión debe introducir una lógica adicional en aquellos métodos Python en dónde se espera que las variables de entrada puedan tener tipos distintos.

  La función en cuestión deberá decidir en tiempo de ejecución como convertir estas variables al tipo de dato C++ correcto de modo que se invoca la función sobrecargada correspondiente en cada caso.

- La forma más habitual de imprimir objetos y variables en la «salida standard» de un programa es empleando el operador «<<» y los objetos «std::cout» y «std::endl» sobre el lenguaje C++.

  En Python, los objetos se muestran por pantalla invocando la función «print», pasando el objeto que desea imprimirse como argumento, o escribiendo sencillamente su nombre (en una sesión python interactiva).

  En ambos casos, el intérprete ejecuta de forma interna un metamétodo definido por la clase del objeto con el nombre «__str__». El valor devuelto por la función (una cadena de caracteres) es imprimida en la salida.

  La extensión provee una implementación para el metamétodo «__str__» en cada clase definida, de modo que cualquier objeto de esta librería puede ser imprimido por pantalla, mostrando información relevante de los mismos.

4.1.2 Módulo de renderización de gráficos o «drawing»

Este módulo de la librería desarrolla la lógica que permite dibujar las primitivas geométricas creadas para describir un sistema mecánico sobre un entorno gráfico, el cual permite al usuario visualizar el mecanismo e interactuar con este.

Las primitivas geométricas representadas como instancias de las clases definidas por el núcleo de la librería, son convertidas internamente a «entidades renderizables». Cada entidad tiene asociada una geometría específica (un modelo 3D o «mesh») que determina su apariencia a la hora de ser dibujada, y una matriz de transformación que permite establecer como se posicionará su geometría sobre la escena 3D.

El modelo 3D asociado a una entidad es un conjunto de vértices conectados mediante aristas. Al renderizar la geometría, se muestran sus caras (subconjuntos de tres o más vértices unidos por un bucle cerrado de aristas).

Los modelos 3D pueden ser generados de forma dinámica cuando su geometría es sencilla (por ejemplo para crear cubos, cilindros, conos, . . .) o importados de archivos con la extensión «.stl» (los cuales pueden ser diseñados por herramientas de software como OpenSCAD, Blender, 3D Max).

La matriz de transformación de una entidad determina como debe proyectarse su geometría sobre el espacio 3D. Puede representar una o más transformaciones afines (como escalado, rotación, traslación, . .) aplicadas en un orden específico.

Las coordenadas de cada vértice son procesadas empleando la matriz de transformación; Por cada vértice se calcula el producto matricial entre sus coordenadas y la matriz, obteniendo de este modo un nuevo vector que representará los nuevos valores para sus componentes en el espacio 3D, una vez aplicada la transformación afín.

Transformación de primitivas a entidades renderizables

Las primitivas empleadas para la definición del sistema mecánico deben convertirse a entidades renderizables para ser dibujadas. Este proceso consta de dos fases:

- Selección de una geometría 3D
- Computo de la matriz de transformación afín

En función del tipo de primitiva, se dibujará empleando una geometría diferente. Las primitivas punto, frame (sistema de referencia) y vector tendrán asociadas formas geométricas básicas (auto-generadas) mientras que los sólidos deberán mostrarse mediante modelos 3D importados de archivos «.stl».

- Se utilizan esferas para representar los puntos.
- La geometría para los vectores es representada con una esfera (en el origen), un cilíndro para mostrar el cuerpo y un cono en el extremo del vector.
- Finalmente, los sistemas de referencia se dibujan con tres vectores (cada uno apuntando a una dirección distinta del espacio 3D).

Esta es la forma en la que se selecciona el modelo 3D para cada componente del sistema mecánico, aunque la librería permite asociar geometrías distintas (utilizando los métodos de la API) a las establecidas por defecto.

A continuación se calcula la matriz de transformación para la entidad. Cada transformación afín (escalado, rotación y traslaciones) puede ser representada mediante una matriz cuadrada de orden 4. Se determina en primer lugar que transformaciones son necesarias (y el orden en el que se aplican) para que la geometría se proyecte de forma correcta en el espacio 3D.

Se computa el producto matricial entre las matrices asociadas a cada una de las transformaciones que deben aplicarse. La matriz resultante representa una concatenación de todas las transformaciones, que se utilizará para proyectar los vértices de la geometría.

Un detalle a tener en cuenta es que los elementos de estas matrices deben ser expresiones simbólicas, ya que se calculan a partir de las propiedades geométricas de las primitivas, que a su vez se definen como fórmulas matemáticas en las que aparecen variables (coordenadas, parámetros, . . .).

Puesto que el valor de estas variables es dinámico (por ejemplo, al ejecutar una simulación del sistema), la matriz de transformación debe ser evaluada de forma numérica (sustituyendo los símbolos por sus respectivos valores) cada vez que sea necesario transformar la geometría.

Las matriz de transformación asociada a una entidad renderizable es llamada «model matrix», pero además se emplean otras dos transformaciones adicionales sobre cada vértice:

- Una proyección sobre las coordenadas del espacio de la cámara. De este modo, se simula estar tomando una captura de la escena con una cámara (posicionada en un lugar concreto del espacio y apuntado en una dirección especifica) Esta transformación tiene asociada una matriz que se denomina «view matrix»
- Finalmente, se aplica una proyección sobre el espacio 2D de modo que las coordenadas de la geometría puedan expresarse en pixeles de la pantalla. La matriz necesaria para llevar a cabo esta operación se llama «projection matrix»

Simulación de un sistema mecánico

Este módulo también implementa un algoritmo que permite realizar simulaciones de un sistema mecánico. Entendemos por simulación una animación sobre el entorno gráfico que permite visualizar el comportamiento dinámico del mecanismo en cuestión.

La simulación puede ser configurada ajustando múltiples parámetros:

- **fps**: Número de refrescos por segundo del entorno gráfico o la frecuencia con la que se actualiza la escena 3D
- **delta_time**: Es el valor de paso para la variable tiempo de la simulación (independiente del parámetro fps).
- **time_limit**: Es la duración total (opcional si la simulación debe continuar de forma indefinida)
- **looped**: Indica si la simulación debe reproducirse en bucle o terminarse cuando llega al tiempo límite.

En cada «paso» de la simulación, se ejecuta el siguiente algoritmo:

- Se actualiza el valor numérico de la variable tiempo utilizando el valor de paso de la simulación.
- Se ajustan los valores numéricos de las coordenadas (y sus velocidades) utilizando las ecuaciones cinemáticas y resolviendo analíticamente las ecuaciones de restricción (está lógica es implementada por el núcleo de la librería).
- Utilizando los valores actuales de las variables del sistema mecánico, evaluar numéricamente las matrices de transformación asociadas a las entidades gráficas 3D de la escena (como se indica en la sección anterior).
- Finalmente, dibujar la escena y refrescar la ventana gráfica.
- Finalizar la simulación si se alcanza el tiempo limite de la misma (volverla a iniciar si está en modo bucle).

Estructura del módulo

Este componente de la librería divide su implementación en varios ficheros de código fuente. En ellos se emplea el paradigma de programación orientado a objetos para definir un conjunto de clases. Entre estas destacamos las más relevantes y que función desempeñan:

- **Drawing**: Representan «entidades renderizables» (es una generalización de las siguientes dos clases).
- **Drawing2D**: Define una «entidad renderizable» en el espacio de coordenadas 2D de la pantalla.
- **Drawing3D**: Es una «entidad renderizable» en el espacio de coordenadas 3D.
- **Geometry**: Representa una geometría o modelo 3D.
- **Transform**: Es una transformación afín que puede ser aplicada sobre las entidades 3D.
- **ComposedTransform**: Representa una secuencia de transformaciones afines que son aplicadas en un orden específico sobre una entidad 3D (es una subclase de la anterior)
- **Simulation**: Describe y permite manipular el estado de la simulación de un sistema mecánico
- **Camera**: Es una entidad no renderizable pero que vive en la escena 3D y representa la cámara que se utilizará para capturar el resto de entidades y mostrarlas en el entorno gráfico.
- **Scene**: Agrupa todas las entidades de la escena: La cámara y el resto de entidades tanto 2D como 3D.
- **Viewer**: Representa una ventana creada por el sistema operativo en la cual se proyecta la escena.

La clase Drawing3D posee además múltiples implementaciones (subclases):

- **PointDrawing**: Es una entidad 3D que permite dibujar una primitiva tipo punto del sistema mecánico.
- **VectorDrawing**: Permite dibujar una primitiva tipo vector.

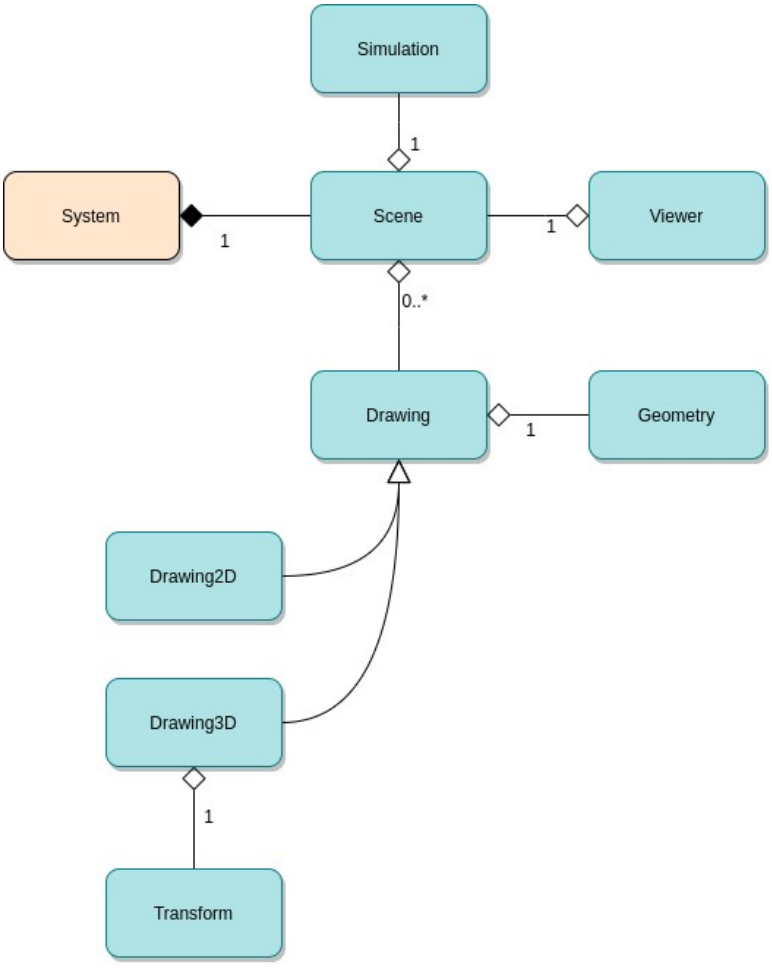
- **PositionVectorDrawing**: Es una especialización de la anterior. La diferencia es que el vector es dibujado partiendo de un origen (definido por una primitiva tipo punto).
- **VelocityVectorDrawing**: Especialización de VectorDrawing para dibujar vectores velocidad.
- **FrameDrawing**: Permite dibujar primitivas tipo «frame» o sistemas de referencia.
- **SolidDrawing**: Entidad 3D para renderizar sólidos.

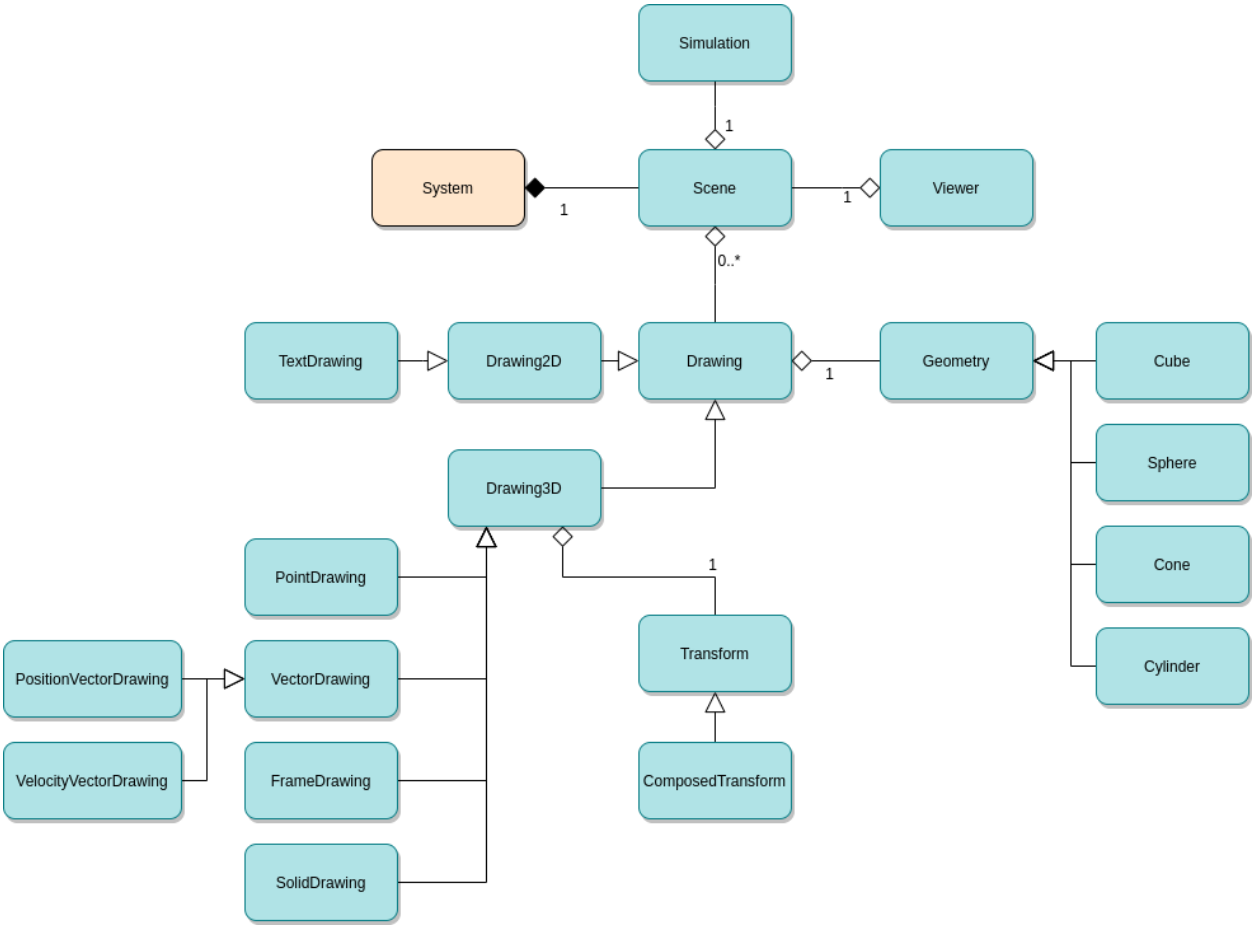
Geometry también tiene clases especializadas para representar geometrías 3D básicas:

- **Cube**: Define una geometría en forma de cubo.
- **Cylinder**: Representa una geometría cilindrica.
- **Cone**: Permite crear geometrías cónicas.
- **LineStrip**: Un objeto de esta clase es una secuencia de puntos interconectada mediante líneas.
- **Sphere**: Permite generar una geometría esférica.

Geometry define además un método para importar modelos 3D desde archivos con la extensión «.stl».

Los siguientes diagramas muestran las relaciones de dependencia (agregaciones, asociaciones y herencia) entre clases:





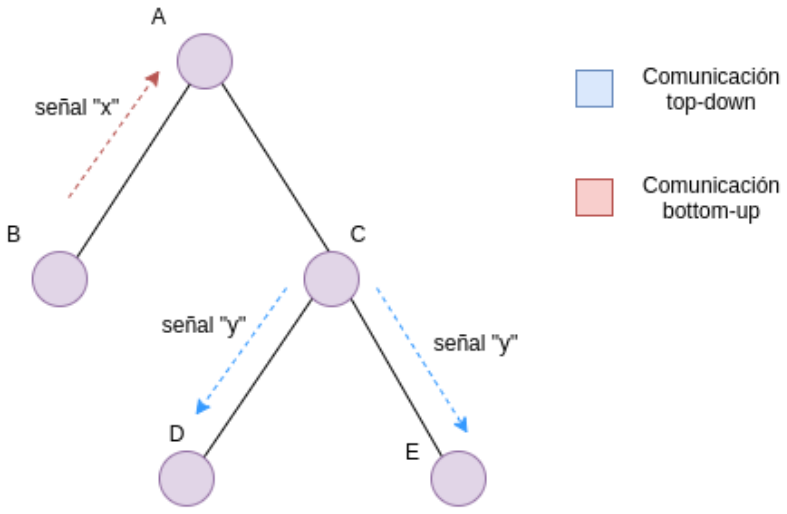
La clase System que aparece en el diagrama no está definida en este módulo (es parte del núcleo de la librería).

Comunicación entre componentes

Las clases de este módulo deben coordinarse entre sí para funcionar de forma correcta. No todas las clases que deben coordinarse tienen una relación de dependencia directa, lo que dificulta la comunicación entre estas; Si un objeto necesita enviar un mensaje a otro, primero debe obtener una referencia al mismo, consultando de forma recurrente sus objetos agregados, y los objetos agregados de los agregados, . . .

Para reducir la complejidad en la comunicación entre objetos, todas las clases de este módulo implementan una interfaz común, de modo que pueda aplicarse el patrón de diseño software «event producer»:

- Con independencia de las relaciones de agregación y herencia de las clases, cada objeto puede verse como un nodo que forma parte de una estructura o jerarquía en forma de árbol. Un nodo puede tener varios nodos hijos y un solo nodo padre (estos también son objetos y pueden ser de cualquier clase).
- Los nodos pueden comunicarse entre ellos empleando dos tipos de mecanismos de paso de mensajes diferentes: Un nodo puede enviar un mensaje de abajo hacia arriba en la jerarquía, de modo que sus nodos antecesores pueden capturarlo y ejecutar una acción específica (comunicación bottom-up). Además de ello, este puede enviar un mensaje a todos sus nodos predecesores, propagándose desde arriba hacia abajo (comunicación top-down)



- La señal x se envía del nodo B al nodo A
- La señal y es enviada por C y se propaga a sus predecesores D y E

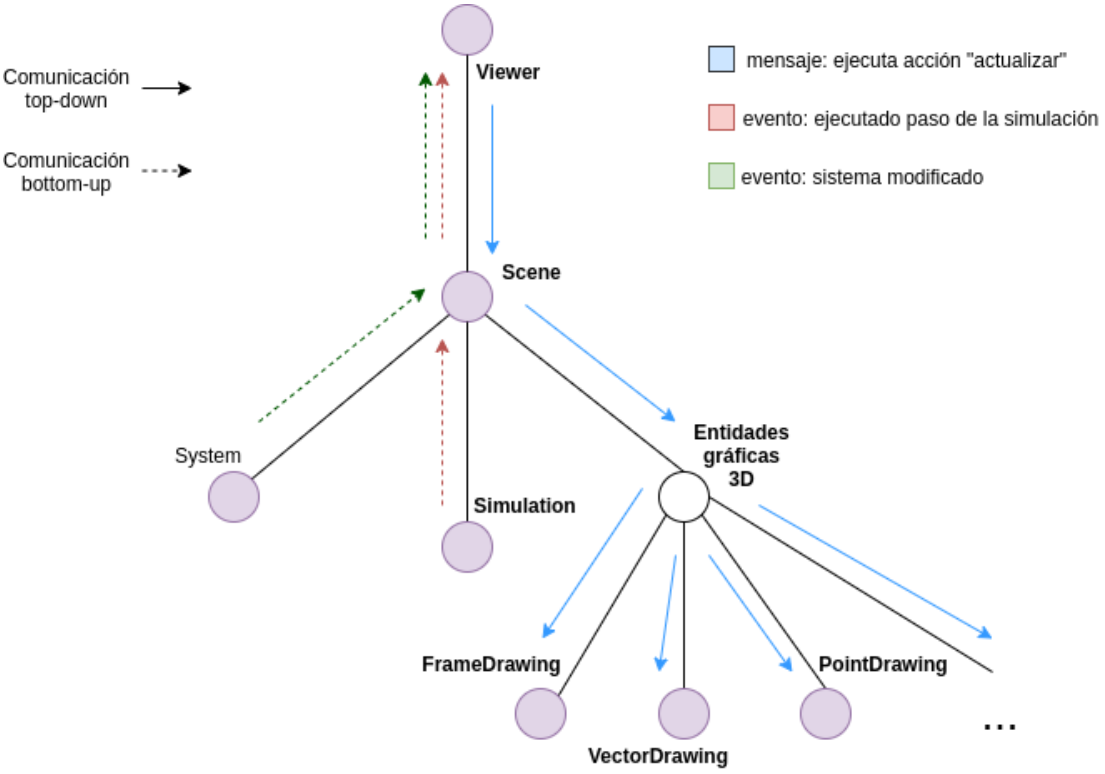
Los mensajes enviados entre nodos pueden clasificarse en dos categorías:

- Peticiones o mensajes activos: Cuando el emisor del mensaje quiere que los receptores del mismo ejecuten una acción específica.
- Eventos o mensajes pasivos: El emisor no espera que el mensaje sea recibido. Suele ser enviado cuando un objeto modifica su estado (estos normalmente se propagan de abajo hacia arriba en la jerarquía). Los receptores en este caso son los que actúan de forma activa, decidiendo la acción a realizar en caso de recibir la señal.

Para orquestar la comunicación de las clases de este módulo, la jerarquía de objetos descrita se estructura del siguiente modo: El visor 3D (instancia de la clase Viewer) es el nodo raíz del árbol. El objeto escena es un nodo hijo del anterior, el cual tiene como predecesores un objeto de la clase Simulation, una instancia de la clase System y las entidades de la escena.

La lógica más relevante en lo que respecta a la comunicación entre estas clases se produce del siguiente modo:

- La clase System (objeto del núcleo de la librería) se encarga únicamente de enviar eventos cada vez que el sistema mecánico se ve alterado de algún modo. Por ejemplo, cuando el valor numérico de una coordenada o parámetro es modificado.
- La instancia de la clase Simulation envía un evento siempre que el estado de la simulación cambie o se ejecute un nueva iteración o «paso» en esta.
- Cuando una nueva entidad 3D es creada y añadida a la escena, la clase Scene genera un evento.
- Cuando una entidad 3D modifica sus propiedades de visibilidad o algún parámetro de su geometría, también se genera un evento.
- La clase Viewer se encarga de recibir todos los eventos lanzados por el resto de objetos. Cuando los recibe, se encarga de refrescar el entorno gráfico. Para ello debe de enviar previamente una señal a todas las entidades 3D para que evalúen de forma numérica sus matrices de transformación.

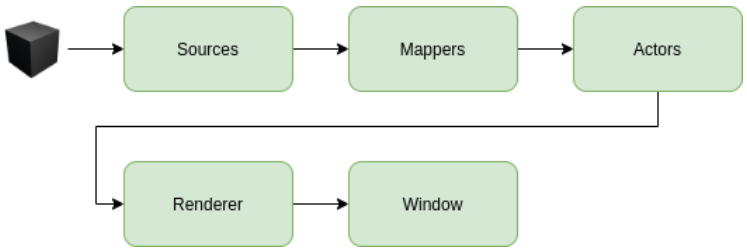


VTK

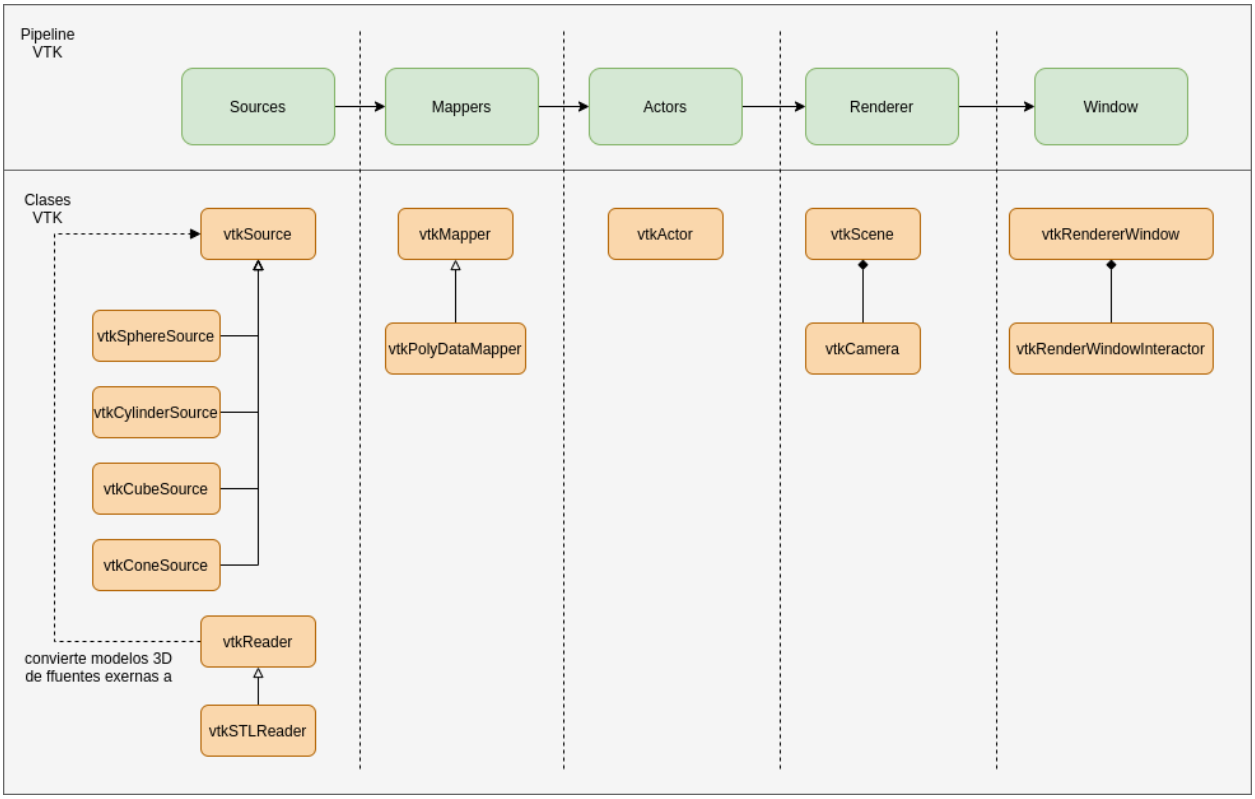
Internamente la librería utiliza el framework de renderización de gráficos VTK (su extensión al lenguaje Python). La implementación de esta librería se ha diseñado de forma semejante a como está estructurada VTK, la cual emplea los siguientes objetos/algoritmos (a nivel conceptual):

- **«sources»** o fuentes, contienen la representación de la geometría 3D a mostrar.
- **«filters»**: Algoritmos que transforman y procesan la geometría de algún modo.
- **«mappers»**: Mapean o convierten la geometría en objetos «tangibles», compuestos por primitivas gráficas que pueden ser mostradas en la escena.
- **«actors»**: Permiten ajustar diferentes parámetros de visibilidad y apariencia de la geometría 3D (color, texturas, . . .).
- **«renderer»**: Controlan el proceso de renderización de la escena.
- **«window»**: Se encargan de mostrar la escena en una ventana gráfica.
- **«interactor»**: Se encarga de gestionar los eventos del usuario capturados por la ventana gráfica.

El proceso desde que la geometría es generada hasta que la escena es mostrada en VTK, se denomina «pipeline» de renderización de gráficos.

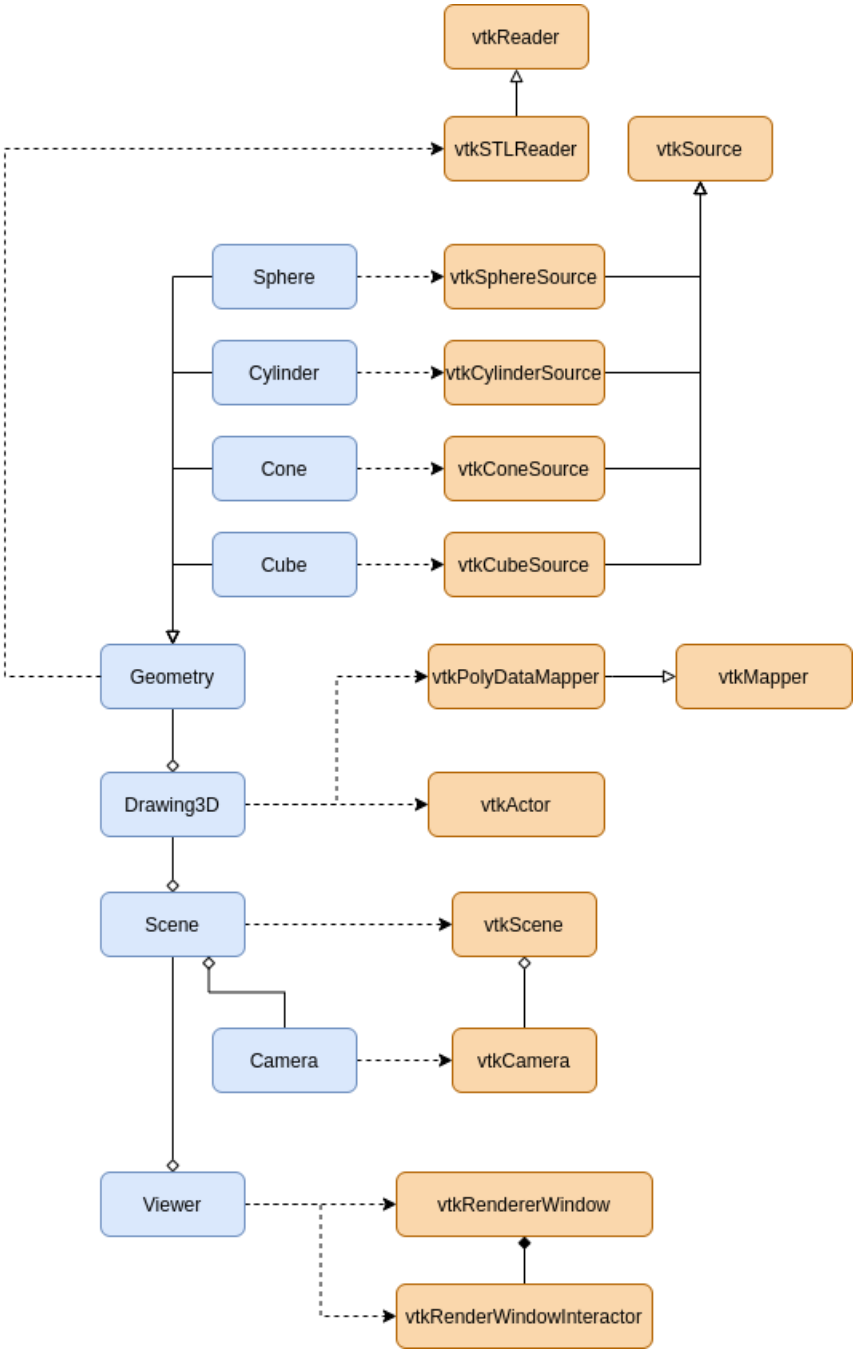


La implementación de VTK en Python emplea el paradigma orientado a objetos. De ellas se utilizan las clases que se muestran en el diagrama:



Cada una de ellas tiene un papel en una fase específica del pipeline de renderización.

La siguiente figura muestra las relaciones de dependencia y uso entre las clases de este módulo y las de VTK:



4.1.3 El módulo de interfaz de usuario

El último componente de la librería añade una interfaz de usuario al visor 3D (desarrollado en el módulo anterior). Su diseño es el mencionado previamente en la sección *Diseño de la interfaz de usuario*

Se desarrolla como una modificación de IDLE (framework para la ejecución interactiva de código Python), que se distribuye como un paquete dentro de la librería standard del lenguaje. IDLE a su vez es creado usando la librería tkinter (librería de creación de interfaces de usuario por defecto de Python).

IDLE ha sido modificado para añadir la opciones del menú contextual e insertar el visor 3D que aparece a la derecha de la ventana.

4.2 API funcional de la librería

La implementación de la librería se ha diseñado con el paradigma de programación orientada a objetos, aunque es posible el empleo del paradigma funcional para hacer uso de los elementos de la API.

«System» es la clase más importante y concentra la mayor parte de las funcionalidades del núcleo de la librería. No suele instanciarse más de una vez en cada sesión Python, ya que en raras ocasiones el usuario define dos sistemas mecánicos en un mismo programa.

Utilizando el paradigma orientado a objetos, es necesario acceder mediante la operación de acceso a atributos del objeto «System» creado para obtener una referencia a los métodos e invocarlos:

```
>>> sys = System()
>>> sys.new_param(...)
>>> sys.new_vector(...)
...
```

La API funcional expone todos los métodos de la clase System como rutinas globales. Cada una de ellas delega la llamada a una instancia de la clase System creada por defecto. De este modo, el código queda más claro y sencillo:

```
>>> new_param(...)
>>> new_vector(...)
```

También expone los rutinas de dibujo de la clase «Scene». Para invocar una función del objeto de esta clase, agregado a una instancia de la clase «System», es necesario obtener una referencia al método del siguiente modo:

```
>>> sys = System()
>>> scene = sys.get_scene()
>>> scene.draw_point(...)
```

La API funcional simplifica la escritura del código anterior y lo reduce a una única instrucción:

```
>> draw_point(...)
```

CHAPTER 5

Pruebas de software

El software de esta librería es verificado para comprobar que funciona de forma correcta. Para ello es sometido una batería de pruebas:

- **Pruebas unitarias**: Sobre las clases de la librería más relevantes, se definen una serie de tests que verifiquen su correcto funcionamiento.

  Cada test valida un método no trivial definido por la clase. Se encarga de comprobar que los valores de salida de la función, son los esperados para ciertos valores de entrada (comprueba que se cumplan las «postcondiciones» del método). Por otro lado, también debe asegurarse que la rutina lanza un error si los argumentos de entrada no satisfacen «precondiciones» de la función (o tengan valores del tipo Python incorrecto).

  Estas pruebas son modulares e independientes, es decir, pueden ejecutarse de forma aislada sin afectar al resultado de otros tests unitarios. Son automatizables ya que no requieren la intervención del programador.

  Cada una de ellas es codificada como un programa Python empleando el framework de pruebas unitarias «pytest».

- **Pruebas de integración**: Estos tests comprueban la interacción de los distintos módulos y componentes de la arquitectura software. Verifica que la comunicación entre todos los elementos de la librería se lleva a cabo de forma correcta.

- **Pruebas de despliegue**: Comprueban la compatibilidad del software sobre distintas plataformas o entornos de producción.

  Son ejecutadas empleando la tecnología Docker, que permite crear de forma procedimental un sistema operativo embebido en un contenedor virtual, sobre el cual se intenta instalar la librería junto con todas sus dependencias.

  Si este proceso tiene éxito, se sabrá que el entorno construido dentro del contenedor es compatible con el software desarrollado. Las pruebas de despliegue se han realizado sobre distintas versiones del S.O Linux Ubuntu y con diferentes versiones del lenguaje Python.

  Algunas de estas pruebas han sido automatizadas mediante los «workflows» o flujos de trabajo de Github, que permiten someter el software a estos tests (ejecutados en la nube por servidores de esta misma plataforma) siempre que se publique una nueva versión «alpha» de la librería.

Las pruebas unitarias y de integración no garantizan al cien por cien el correcto funcionamiento del software, ya que:

- La cobertura de código de los tests no es completa (aunque abarca las partes más importantes)-

- Muchas de las funcionalidades implementadas por el núcleo de la librería son delegadas a las librerías C++ lib_3d_mec_ginac y GiNaC. Cabe la posibilidad entonces de que estas contengan errores a pesar de que la interfaz Python funcione adecuadamente.

CHAPTER 6

Documentación del proyecto

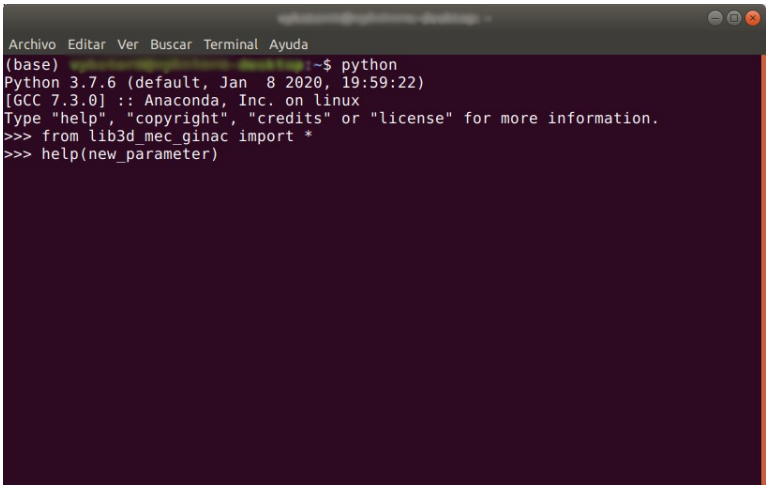
El objetivo de la documentación es dar a conocer las funcionalidades software desarrolladas, además de facilitar las tarea de aprendizaje y utilización de la librería a los usuarios finales.

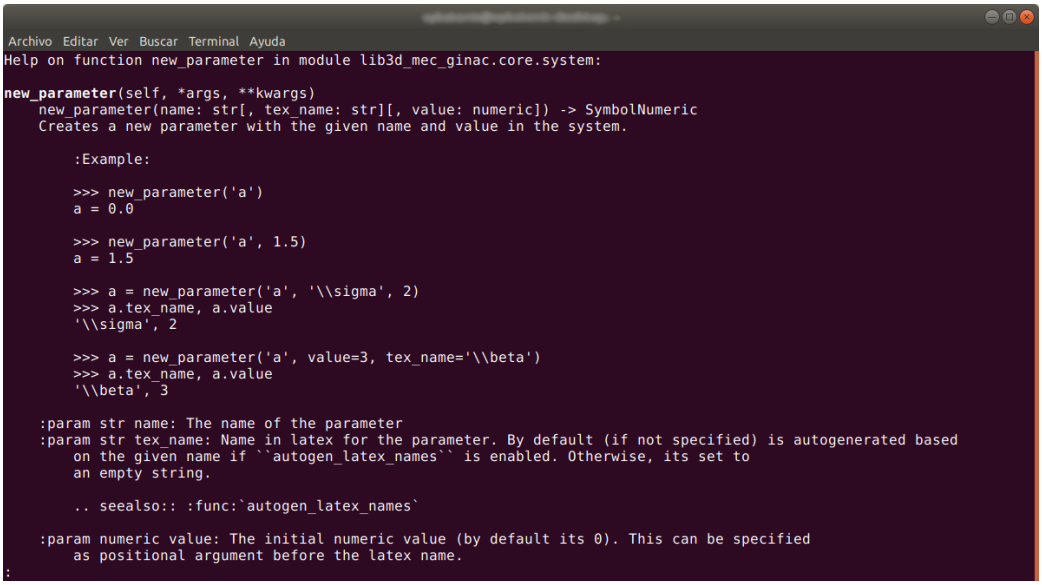
La documentación del proyecto se presenta en distintos medios:

- Información técnica sobre las clases y rutinas de la API. Se describe cual es la función de cada elemento de la interfaz y como debe emplearse. Esta tipo de documentación es denominada «API reference» y es accesible de varias formas:
  - En una consola Python interactiva, utilizando la función help, es posible obtener la documentación técnica de una clase o función específica de la API.
  - En la siguiente página web, se muestra también la documentación de la API al completo en formato HTML.
- README.md: Es un fichero de texto localizado en el repositorio de código del proyecto que contiene la información básica para el uso e instalación de la librería.
- Por último y no menos importante, esta memoria es un recurso documental que permite a los lectores tener una idea básica de la utilidad del software creado y como se ha desarrollado.

6.1 Referencia de la API

Los elementos de la API se han documentado en el código fuente añadiendo comentarios en sus definiciones, llamados «docstrings». La función *help* de Python muestra la documentación almacenada en los «docstrings».





La referencia de la API está escrita en el lenguaje de marcado «reStructuredText», que es convertido a páginas HTML o archivos pdf mediante la herramienta Sphinx.

Está última es empleada junto con la extensión «autodoc» que permite importar los docstrings de las funciones a la documentación para darles formato y estilo, como se muestra en la siguiente figura:

new_parameter(*name: str[, tex_name: str][, value: numeric]*) → SymbolNumeric

Creates a new parameter with the given name and value in the system.

Example:

```
>>> new_parameter('a')
a = 0.0
```

```
>>> new_parameter('a', 1.5)
a = 1.5
```

```
>>> a = new_parameter('a', '\\sigma', 2)
>>> a.tex_name, a.value
'\\sigma', 2
```

```
>>> a = new_parameter('a', value=3, tex_name='\\beta')
>>> a.tex_name, a.value
'\\beta', 3
```

Parameters:
- **name** (*str*) – The name of the parameter
- **tex_name** (*str*) –
  Name in latex for the parameter. By default (if not specified) is autogenerated based on the given name if `autogen_latex_names` is enabled. Otherwise, its set to an empty string.

  See also: `autogen_latex_names()`
- **value** (*numeric*) – The initial numeric value (by default its 0). This can be specified as positional argument before the latex name.

Return type: SymbolNumeric

Raises:
- **TypeError** – if the given arguments have incorrect types

CHAPTER 7

Publicación del software

7.1 El código fuente

El código fuente del proyecto está disponible al público en un *repositorio alojado en el servicio online Github <https://google.com>*. Puede descargarse manualmente desde su página web o utilizando directamente la interfaz por línea de comandos de git en una sesión bash de Linux.

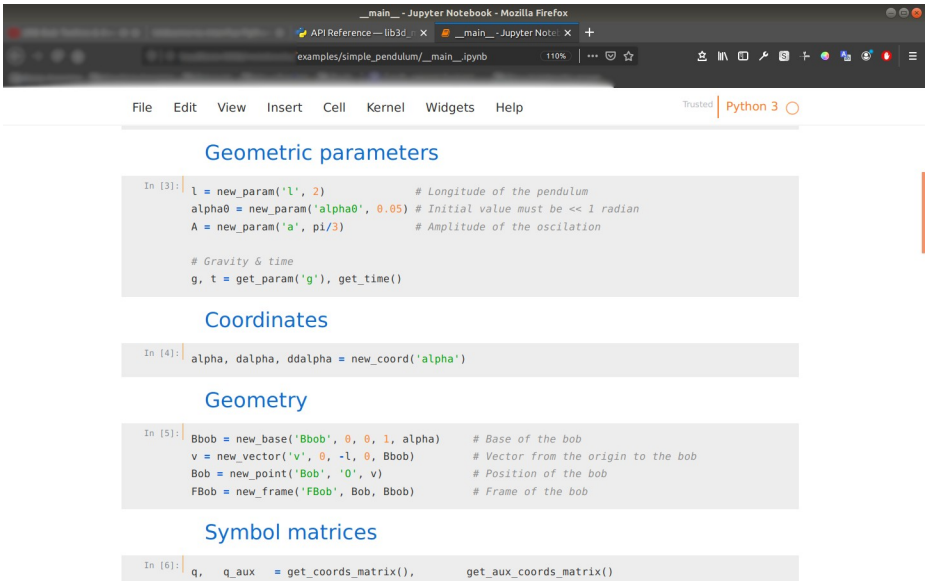
Git permite mantener distintas versiones del código fuente del proyecto, creando múltiples ramas de código. Esto permite mantener el «entorno de producción» (versión del software estable o versión «alpha») separada del «entorno de desarrollo» (última versión del software donde se prueban nuevas funcionalidades, o versión «beta»).

7.2 Imágenes docker

El software puede obtenerse como una imágen docker (entorno virtual) la cual contiene la versión de sistema operativo Linux Ubuntu 18.04 junto con esta librería y todas sus dependencias ya instaladas. Los usuarios pueden descargar la imagen y utilizarla de modo que no es necesaria la fase de instalación. Esta imagen está disponible en el servicio de paquetes de Github, accesible desde el repositorio del proyecto.

7.3 Servicios en la nube

También se ha puesto a disposición del usuario una página web en donde es posible utilizar el software del proyecto de forma online. La página (gestionada por el servicio «Heroku») es un notebook de «jupyter»; Un documento que permite entremezclar código Python, texto, imágenes y otros recursos.



7.4 Licencia

El proyecto queda bajo los términos de la licencia GNU General Public License (GPL) versión 2. Esto significa que puede utilizarse para aplicaciones comerciales, modificarse y distribuirse. Si se modifica, deberá incluirse el código fuente y contener una copia de la misma licencia.

La razón por la cual se ha escogido GPLv2, es debido a que las librerías C++ «lib_3d_mec_ginac» y «GiNaC» están licenciadas también bajo las mismas condiciones y porque sus archivos binarios y parte de su código fuente son incluidos en el repositorio.

CHAPTER 8

Conclusiones y líneas futuras

En mi opinión, todos los objetivos establecidos del proyecto se han conseguido con éxito y se han utilizado los patrones de diseño software adecuados en cada módulo de la librería y adaptados a la idiosincrasia del lenguaje y el ecosistema Python. La integración de la librería con otras herramientas software conocidas como Numpy (librería de computación numérica) y VTK (framework de visualización) añaden además mayor calidad al producto final.

La posibilidad de utilizar la librería sobre distintos entornos programáticos (sesiones interactivas Python, la interfaz de usuario y los «notebooks de jupyter» online) es un punto adicional en cuanto a la usabilidad y accesibilidad del software.

Se puede decir en definitiva que se ha cumplido con el metaobjetivo del proyecto, el cual era facilitar la interacción entre el usuario y las funcionalidades de la librería «lib_3d_mec_ginac».

Por otro lado, cabe destacar algunas dificultades encontradas durante el desarrollo del proyecto y cómo se podría haber mejorado:

- La elección del framework para la creación del «language binding» o extensión de C++ a Python no fue la más óptima. Se escogió Cython for su facilidad de uso, pero podría haberse utilizado «Boost.Python» para esta misma tarea, ya que existía un librería (pynac) elaborada con esta última para hacer interface a algunas partes del software de computación simbólica en C++ GiNaC. De haber empleado «Boost.Python», el desarrollo de la extensión habría sido más rápido.
- VTK es un software de visualización robusto y completo pero la mayor parte de las capacidades que ofrece no han sido utilizadas. Esta además ha dado bastantes problemas de compatibilidad con distintas versiones del intérprete Python. Podría haberse empleado un software de renderización de gráficos de Python más ligero como PyOpenGL o PyGame.

También se han dejado algunas mejoras por implementar de cara al futuro:

- Agilizar la fase de instalación de la librería. La extensión Cython se compila «in situ» durante la instalación, haciendo que esta fase sea más lenta. Además obliga al usuario a disponer de un compilador C++ (con soporte para la versión 11 o superior del lenguaje).

  En un futuro, la extensión podría distribuirse ya precompilada de modo que no sea necesaria su compilación en la máquina del usuario.

- Realizar más pruebas de despliegue sobre otros sistemas operativos. Solo se ha probado hasta la fecha el funcionamiento de la librería sobre la plataforma Linux Ubuntu. Se desea comprobar si esta puede ser utilizada en otros sistemas basados en Unix.
- Añadir más ejemplos de uso en el repositorio que modelen sistemas mecánicos más complejos. Por el momento solo se dispone de ejemplos triviales (mecanismo de cuatro barras o «four bar linkage» y un péndulo simple)

Cómo reflexión final, a nivel personal el desarrollo de este proyecto a expandido mis conocimientos sobre los lenguajes C++ y Python (y todas las librerías utilizadas). Hasta la fecha desconocía el área de la computación simbólica y los sistemas de álgebra por ordenador (CAS). Ha sido un reto familiarizarme con la terminología y los fundamentos de esta área del conocimiento, además de aprender a utilizar el framework «GiNaC».

Por último, he disfrutado descubriendo los conceptos de la teoría de sistemas mecánicos dinámicos multicuerpo y su conexión con el álgebra simbólica, a medida que he ido desarrollando la interfaz lib_3d_mec_ginac a Python.

Espero que el software de este proyecto sirva como ayuda y herramienta a los ingenieros mecánicos del futuro para diseñas sistemas de forma virtual que luego puedan tener alguna utilidad práctica en el mundo real, y al personal docente de las universidades para enseñar conceptos relacionados sobre la mecánica multicuerpo.

CHAPTER 9

Bibliografía

Para el desarrollo del proyecto se han recurrido con frecuencia a la documentación (referencia y tutoriales) de los lenguajes de programación empleados:

- **Cython**: https://cython.readthedocs.io/en/latest/
- **Python**: https://docs.python.org/3/
- **C++**: http://www.cplusplus.com/reference/
- **reStructuredText**: https://www.sphinx-doc.org/es/master/usage/restructuredtext/index.html

También sobre las distintas librerías/tecnologías utilizadas:

- **GiNaC**: https://ginac.de/
- **Sphinx**: https://www.sphinx-doc.org/es/stable/markup/inline.html
- **autodoc** (extensión de sphinx): https://www.sphinx-doc.org/es/master/usage/extensions/autodoc.html
- **NumPy**: https://numpy.org/doc/stable/reference/index.html
- **VTK**: https://vtk.org/doc/nightly/html/
- **Docker**: https://docs.docker.com/get-started/

Y por último los siguientes papers:

- Javier Ros, Aitor Plaza, Xabier Iriarte, and Jesús Pintor. *Symbolic multibody methods for real-time simulation of railway vehicles. Multibody System Dynamics*. 06/2017.
- Javier Ros, Javier Gil, and Isidro Zabalza. *3D_MEC: An application to teach mechanics*. 01/2005.

CHAPTER 10

Apéndice

10.1 Operaciones aritméticas entre objetos

Las siguientes tablas muestran las operaciones disponibles y el resultado obtenido en función del tipo de los operandos.

En la primera fila y columna se indica el tipo de operandos (clase Python). El resto de celdas indican el resultado obtenido (también una clase Python). Si la operación no está implementada, se indica con el signo «—»

cte hace referencia a valores de tipo numérico (real o entero). En Python, los valores reales son representados por la clase `float` y los enteros por la clase `int`

- Operación suma y resta:

+/-	cte	symbol	expr	matrix	vector	tensor	wrench
cte	cte	expr	expr	—	—	—	—
symbol	expr	expr	expr	—	—	—	—
expr	expr	expr	expr	—	—	—	—
matrix	—	—	—	matrix	—	—	—
vector	—	—	—	—	vector	—	—
tensor	—	—	—	—	—	tensor	—
wrench	—	—	—	—	—	—	wrench

- Operación de multiplicación:

•	cte	symbol	expr	matrix	vector	tensor	wrench
cte	cte	expr	expr	matrix	vector	tensor	wrench
symbol	expr	expr	expr	matrix	vector	tensor	wrench
expr	expr	expr	expr	matrix	vector	tensor	wrench
matrix	matrix	matrix	matrix	matrix	—	—	—
vector	vector	vector	vector	—	expr	—	—
tensor	tensor	tensor	tensor	—	vector	tensor	—
wrench	wrench	wrench	wrench	—	—	—	wrench

- Operación de división:

/	cte	symbol	expr
cte	cte	expr	expr
symbol	expr	expr	expr
expr	expr	expr	expr
matrix	matrix	matrix	matrix
vector	vector	vector	vector
tensor	tensor	tensor	tensor
wrench	wrench	wrench	wrench

- Operación potencia (base ** exponente):

^	cte	symbol	expr
cte	cte	expr	expr
symbol	expr	expr	expr
expr	expr	expr	expr